TURING 图灵程序设计丛书

MANNING



Erlang and OTP in Action

Erlang/OTP 并发编程实战

[美] Martin Logan
Eric Merritt 著
[瑞典] Richard Carlsson
连城 译

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Martin Logan

从1999年开始活跃于Erlang社区，后来全职从事Erlang研发。目前任职于全球最大的在线旅游公司Orbitz Worldwide，为基于大规模分布式服务的基础设施开发解决方案。Erlware联合创始人，核心开发人员，Erlang/OTP软件包管理系统Faxien的主要开发者。

Eric Merritt

专注并发编程和分布式系统。曾任职于Amazon.com，现为eCD Market软件工程师。Erlware联合创始人，Erlang芝加哥用户组核心成员。Erlware团队开源产品核心开发人员，Erlang/OTP构建系统Sinan的主要开发者。

Richard Carlsson

瑞典乌普萨拉大学高性能Erlang计划（HiPE）早期成员，研究Erlang技术达17年，曾为标准库、Erlang编译器、运行时系统和Erlang语言本身都作出过不少贡献。此外，他还是Erlang文档系统EDoc和单元测试框架EUnit的创建者。目前加入了Kreditor，致力于高可用性支付系统的Erlang开发。

连城

百度资深软件工程师，《Erlang并发编程（第一部分）》社区翻译项目组织者及主要译者。对分布式存储、分布式消息系统、程序语言设计实现抱有浓厚兴趣。

图灵程序设计丛书

Erlang and OTP in Action

Erlang/OTP并发编程实战



[美] Martin Logan
Eric Merritt 著
[瑞典] Richard Carlsson
连城 译

人民邮电出版社
北京

图书在版编目（ＣＩＰ）数据

Erlang/OTP并发编程实战 / (美) 洛根 (Logan,M.),
(美) 梅里特 (Merritt,E.),(瑞典) 卡尔森(Carlsson,R.)
著 ; 连城译. -- 北京 : 人民邮电出版社, 2012.7
(图灵程序设计丛书)
书名原文: Erlang and OTP in Action
ISBN 978-7-115-28559-1

Ⅰ. ①E… Ⅱ. ①洛… ②梅… ③卡… ④连… Ⅲ. ①
程序语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第122366号

内 容 提 要

本书侧重生产环境下的 Erlang 开发，主要讲解如何构建稳定、版本控制良好、可维护的产品级代码，凝聚了三位 Erlang 大师多年的实战经验。

本书主要分为三大部分：第一部分讲解 Erlang 编程及 OTP 基础；第二部分讲解如何在实际开发中逐一添加 OTP 高级特性，从而完善应用，作者通过贯穿本书的主项目——加速 Web 访问的分布式缓存应用，深入浅出地阐明了实践中的各种技巧；第三部分讨论如何将代码与其他系统和用户集成，以及如何进行性能调优。

本书面向 Erlang 程序员，以及对 Erlang/OTP 感兴趣的开发人员。

图灵程序设计丛书

Erlang/OTP并发编程实战

◆ 著　[美] Martin Logan [美] Eric Merritt
　　[瑞典] Richard Carlsson
　译　连　城
　责任编辑　毛倩倩
　执行编辑　刘美英

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
　邮编 100061　电子邮件 315@ptpress.com.cn
　网址 http://www.ptpress.com.cn
　北京　　　印刷

◆ 开本：800×1000 1/16
　印张：22.25
　字数：526千字　2012年7月第1版
　印数：1-4 000 册　2012年7月北京第1次印刷

著作权合同登记号 图字：01-2011-0610号

ISBN 978-7-115-28559-1

定价：79.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版权声明

译　者　序

本科毕业后，我的第一份工作是即时通信服务研发。大约在2007年年底的时候，出于工作原因，我对XMPP产生了兴趣。在调研过程中，我找来了各种 XMPP服务器进行比较。令我惊讶的是，业内公认最为优秀的分布式XMPP服务器ejabberd，竟然是用一种样貌诡异的冷僻语言写成的——这便是我与 Erlang的第一次碰面。相较于日常惯用的C++和Java，Erlang对我来说既陌生又怪异。囫囵吞枣地过了一遍相关资料之后，我惊讶地发现这门语言竟然已有近三十年的历史，而且功能完备、羽翼丰满。然而，晦涩的语法和文档却令我晕头转向，加上初识函数式语言，思维方式一时难以转变，这第一次亲密接触没过多久便宣告结束。

一年多之后，Facebook发布了基于XMPP的即时通信服务Facebook Chat，所用的服务器正是经过定制的ejabberd[①]。刚好那段时间正在琢磨分布式一致性相关的问题，迫切需要一门便于实现分布式算法的语言。于是我开始在官方文档和*Concurrent Programming in Erlang*的指引下学习Erlang。很快我便发现，只要适应了尾递归和单次赋值等函数式编程的特点，Erlang语言本身非常地简单明了。经过初步的摸索，对Erlang的认识也逐渐完整起来。最初以为这是一门阳春白雪的学院派语言，后来却发现大错特错。自打诞生之日起，Erlang就是一门目的性和工程性极强的语言。它的特性集合历经电信行业的千锤百炼，几乎不带一丝一毫的水分。尤为有趣的是，正如本书简介中所述，Erlang的历史与工程型语言的另一典范C惊人地相似。但是，由于思想过于前卫，这门优秀的语言却一直未能受到足够的关注。在服务器端的开发者们都在兴致勃勃地探讨如何通过避免内存复制来提高单机性能时，复制式消息传递简直就是异端邪说！过去十年间，并发处理的复杂性已经随着硬件瓶颈的到来而凸显出来。然而，对于一线应用的开发者而言，近年来大规模互联网应用的爆炸式增长才切实将并发处理变成了一个亟待解决的现实问题。

没想到很快冷水便劈头盖脸地浇了下来——Erlang的并发、容错机制只是整个体系的基石，要想真正发挥出它们的威力，还要仰仗一套叫做OTP的东西。这玩意儿可真是折腾死人了！OTP行为模式迷宫般的回调逻辑把我绕得晕头转向；在解决具体问题时，我总是搞不清楚到底该用`gen_server`、`gen_fsm`还是`gen_event`；好不容易在Erlang shell里跑通一段代码，想打包部署到其他机器上做多机实验时又撞了一头包：应用、发布镜像、变幻莫测的配置项、漫天飞舞的版本号、语焉不详的官方文档……所有这些魔鬼般的细节无不蚕食着我所剩不多的耐心。

译完本书之后再回过头来看，其实OTP的核心概念并不复杂，当年最让我搓火儿的还是实战

① 参见http://blog.process-one.net/facebook_chat_supports_xmpp_with_ejabberd/。

过程中各种剪不断理还乱的繁琐细节。Erlang/OTP的官方文档并不缺细节，但这些细节却水银泻地般地散落在各个角落里，缺乏一条将它们有机地贯穿起来的主线。梳理这一主线，正是本书的任务。全书以交付产品级代码为目标，以一个现实而鲜活的虚拟项目为舞台，深入浅出地讲解了OTP中最为重要的机制和概念，并清晰地呈现了它们之间的内在联系。如果说 Erlang/OTP的官方文档是个繁华的大都市，那么这本书就是一本地图。明白了关键机制和概念之间的内在联系之后，按图索骥深入学习Erlang /OTP就不再是什么难事了。

既然是译者序，那么再说说译书的那些事儿吧。2009年至2010年间，我在Erlang China社区内发起了CPiE-CN项目，召集了一批志愿译者共同完成了*Concurrent Programming in Erlang* (*Part I*)中文版《Erlang并发编程（第一部分）》的翻译。后来，正是这一项目促使我成为了本书的译者。在此我要对CPiE-CN的几位志愿译者表示感谢，他们是王飞、赵宇坤、张驰原、丁豪、赵卫国和吴峻。在本书近一年的翻译过程中，我要感谢图灵教育的傅志红老师、李松峰老师和刘美英老师的帮助，感谢他们容忍了我非常规的交稿方式（以及蜗牛般的进度）。我还要感谢几位协助校对部分中间译稿的早期读者，他们是赵卫国、田中博和倪华杰。最后，特别感谢我的妻子雅莉：这一年中，本职工作的繁重程度远远超出了我的想象，所剩不多的业余时间完全被这本书消耗殆尽，如果没有她的支持和监督，身为重度拖延症患者的我也许根本就坚持不下来。

好啦，闲话少说，预祝各位读者在享用本书的过程中玩儿得开心！

序

长久以来，Erlang界仅有一本书流传于世，即1993年出版、1996年修订的“红宝书”①。花上100多美元Erlang的拥趸现在还能购买到这本书的印刷版。“红宝书”出版已逾十年，书中的内容早已过时。这门语言几经演变，又新增了一些强有力的编程结构。广泛应用于现代Erlang程序的高阶函数、列表速构（list comprehension）②和比特位语法（bit syntaz）等，“红宝书”都未曾收录。不过，发布于1996年的Erlang应用开发框架——开放电信平台（OTP，Open Telecom Platform）——才是全书空缺内容中最关键的一环。Erlang并不难学，OTP却恰恰相反。像本书作者Martin Logan这种自1999年便开始接触Erlang的早期用户，基本上只能靠不断试错硬碰硬地学习OTP。

在过去的几年中，大量Erlang相关图书不断出版，足以证明语言本身的魅力。我们曾获悉还有几本新书即将问世，其中最受瞩目的便是Martin Logan、Eric Merritt和Richard Carlsson合著的这本书。如今它终于面世了。

我是从1993年开始接触Erlang编程的，那时我正在阿拉斯加的安克雷奇设计灾难应急系统。我购买了一套随QIC磁带发布的HP-UX预编译版Erlang。当时的Erlang规模比现在小，支持库的数量也少。我不得不自行设计数据访问结构、数据库管理器、协议解析器以及错误处理框架——不过我却醉心于此。那时候跟现在可没法比：随着同一年Mosaic浏览器的发布Web才刚刚兴起，开源的概念也还得再过五年才会为人所知。在这样的背景下，要想获取一套支持分布式计算和容错的编程框架，就只能狠狠地砸钱砸时间。我淘遍了市面上所有的相关工具，自认已经对各种商业方案了如指掌。当时的Erlang既生涩又不起眼，语法怪异、文档奇缺，但相较于其他工具，其核心理念却显得更为靠谱。

三年后，我已身处瑞典，就职于爱立信并担任史上最大的Erlang项目的首席设计师。我们正打算用Erlang构建传说中的电信级ATM交换系统，以及一套名为开放电信平台的全新框架。之所以采用这个名字，主要是为了迎合公司老大们的胃口——电信是我们的核心业务；开放是时下的流行词儿；主流观点又认为要想构建一套健壮的复杂系统，你就必须拥有一套用于解决冗余、远程配置支持、在线软件升级以及实时追踪调试等问题的平台。

---

① Joe Armstrong、Robert Virding、Claes Wikström和Mike Williams编写的*Concurrent Programming in Erlang*（Prentice Hall, 1993, 1996）。（另一本Erlang经典图书是Joe Armstrong编写的*Programming Erlang*，人称“Erlang圣经”，中文版《Erlang程序设计》已经由人民邮电出版社出版。——译者注

② 速构（comprehension）这一概念源自ZF集合论中的ZF速构，同时也是函数式编程语言中的一种常见语法结构。Erlang中的速构是一种在现有列表/位串上应用若干约束条件后重新构造新的列表/位串的方法。——译者注

开发工具销售并非爱立信的正业，但早在20世纪70年代早期，这儿就开始设计一些用于满足特定需求的编程语言。值得称道的是，爱立信于1998年开源了Erlang/OTP（当然也有出于自身利益的考虑）。于是全世界的爱好者得以投身其中。起初主要用于电信业，后来又逐步渗入到了其他领域。90年代的时候，我们曾试着向Web开发者们大力推荐Erlang，但那时的Web开发者所面临的挑战并非冗余、可伸缩、高响应度的电子商务站点的构建；这类系统的时代尚未来临，并发也尚未入得主流程序员的法眼。那时，并发是众所周知的难点，是人人都唯恐避之而不及的东西。既然如此，人们又何苦选用一种连写“hello world”都得牵扯上并发的语言呢？

随着Web的爆炸式增长和交互性渐强的Web应用的涌现，冷宫中的Erlang终于被解救出来。物理定律也出人意料地向我们伸出援手——通过提高CPU时钟频率来制造更快的单核芯片的技术终于达到了极限。硬件制造商们打出“免费午餐已经结束”的口号，促使开发者放弃对高速单核处理器的依赖，转而探索如何让程序扩展到多个较弱的核上。这对Erlang来说是绝佳的机遇。这意味着程序员中的许多佼佼者至少会开始注意Erlang，并去思考是什么令它如此特别。大部分人只会瞅上一眼，另一些人则会用自己熟悉的语言去模拟Erlang的理念。这是件好事，它意味着知晓并钟爱Erlang及其背后原理的人将更受市场的青睐。

目前OTP已经在电信以外的几个领域得到验证，那些学习并掌握了它的人无不对它赞不绝口。Erlang/OTP是一个非常强大的平台，但需要花时间来学习。在全新项目中应用它时就更是如此。有意思的是，往往那些在OTP项目中摸爬滚打多年的程序员，也不清楚该如何从头构建一个基于OTP的系统。这是因为应用开发者只需接触整个框架的一小部分。这恰恰是我们在大型项目中追求的目标；但小型初创公司的老板不能指望会有人挑灯夜战，逐个儿搞定OTP发布处理的细碎问题和其他犄角旮旯里的各种头疼事儿，所以必须有一套行之有效的示例和教程。

我们正迫切需要一本关于OTP的好书，而本书的出现正填补了这一空白。Martin Logan、Eric Merritt和Richard Carlsson都拥有大量Erlang实践经验，而且都为Erlang社区作出过杰出的贡献，合著本书可谓“强强联合”。我相信此书定会加速推动Erlang实用化的热潮。

尽请赏析！

Ulf Wiger

Erlang Solution 有限责任公司CTO

前　言

本书试图提炼出成就一名专业Erlang程序员所需的最关键的知识，借此我们才能让这门高效的语言发挥出其最大的潜力。Erlang/OTP功能强大，但直到目前为止，对初学者来说，通过研读OTP文档来自学OTP框架仍然是件令人望而生畏的事情（这些文档探究了很多细节，却缺乏全局观）。

本书三位作者长期从事Erlang相关工作，但各自的发展轨迹却很不一样。

Martin："我是在自己第一份'真正'的工作中接触到Erlang的。此前我一直从事C和C++开发，也有意思得很。我的第一任老板Hal Snyder甚至在20世纪90年代便对多线程深恶痛绝，后来邂逅了Erlang。我那时还只是个实习生，于是他给了我一个要用Erlang完成的项目。原因嘛，嗯，无非是我的工钱低，即便我搞砸了，公司在这桩买卖上的损失也就不过70美元。最后我并没搞砸，我自己写了一个1000行的监督进程，代码不好看，因为那时候我压根儿不知道OTP是什么东西，手边当然也不会有相关的书。在这个过程中，我爱上了这种'靠谱'的开发后端系统的方法，也爱上了Erlang。Erlang给了我洞悉未来的机会：我所写的复杂分布式系统，所用的高级算法，都是我那些使唤着命令式语言的同事们所梦寐以求的，不花上两年工夫码上一百万行代码他们压根儿实现不出来。读了数千页文档，写了数万行代码之后，我依然钟情于它。一路走来，我遇到了许多杰出的人物，能与其中的两位共同撰写本书令我激动不已。2004年我在ACM会议上发言时遇到了Richard，四年后我又遇到了Eric，并和他一同创建了Erlware——一个仍在蓬勃发展中的项目。多年来，Erlang在我的职业生涯和个人生活中一直扮演着重要的角色，今后也仍会如此。"

Eric："我研究Erlang完全是无心插柳。我曾想写一款大规模多人游戏。然而我明白，仅凭一人之力，即便是才华横溢，也干不完所有图形处理的活儿。于是我决定集中精力主攻游戏逻辑部分，觉得借助于合适的工具和语言我应该能解决这部分问题。在我的设计中，我喜欢在游戏中设立多个代理对象，每个代理对象都能随时间自主学习并独立而并发地行动。那时我能想到的唯一可行的办法，就是把每个代理对象都建模为某种并发的东西，但当时我还不知道这个东西是什么。我所掌握的语言没法让一个开发者单枪匹马拿下这么一款游戏。于是，我开始考察各种语言，前前后后一共花了大概五年时间，作了一些深入的研究。我很早就见识过Erlang，虽然很喜欢它的并发特性，但实在受不了它的语法和函数式特质。直到考察了很多编程语言之后，我才重新开始欣赏Erlang并用它编写代码。那款游戏我一直也没能写成，但我确信选择Erlang没有错，经过深入的研究和剖析，我发现这门语言在许多方面都大有用武之地。这大概是2000年或2001年的事情

了。后续几年间，我又自学了OTP。后来，在2005的时候，我在Amazon.com引入了Erlang，发布了Sinan的第一版，还认识了Martin Logan，并和他一起创办了Erlware。2008年，我搬到了芝加哥，开始写书并尽心打理Erlware项目。”

Richard：“我大概是在1995年前后接触Erlang的，那时我正在乌普萨拉大学为计算机科学硕士论文选题。这引导我后来成为高性能Erlang研究组的一名博士研究生，就Erlang编译器和运行时系统做了若干年的研究。我在瑞典和美国的会议上结识了Martin Logan和Eric Merritt，他们对Erlang的热情令我印象深刻，尽管那时候Erlang还是一门鲜为人知的语言——尤其是在美国。在攻读博士学位期间，我搞了几个业余项目，其中语法工具库和EDoc应用都源自我在编译器方面的研究成果，而EUnit原本是为了检查我的学生们的并发编程作业是否符合规范而设计的。走出学术界之后，我做了几年和Erlang无关的工作，基本上都是在用Python、Ruby和C++写程序。不过最近，我加盟了瑞典最成功的一家创业公司，再次全职投入Erlang，目前正投身于高速发展的高可用性支付系统领域。”

我们三人努力从共同的经验中提炼出尽可能多的内容，以便让你在迈向大师级Erlang程序员的道路上少走弯路；我们也希望能借助本书，最终让OTP框架成为每个Erlang程序员——而不是少数能将手册翻烂的人——都能掌握的东西。

致　　谢

我们首先要感谢的是这个项目的牵头人Bob Calco，没有你就不会有这本书，我们希望这本书能让你满意。

感谢所有购买了预览版的读者，感谢你们耐心等待本书的完成定稿前我们差不多把这本书重写了三遍，是你们的热情助我们度过了难关。

感谢Jerry Cattell复查Java代码，感谢Francesco Cesarini对我们的帮助和为本书所做的宣传，感谢Ulf Wiger为本书作序，感谢Kevin A. Smith的驱动程序示例代码，感谢Ryan Rawson在Java和HBase方面提供的帮助，感谢Ken Pratt进行技术校对，还要感谢Alain O’Dea及其他所有对预览版给出反馈意见的读者。

这里要特别感谢各位审稿人，他们在写作过程的各阶段对本书手稿提出了宝贵意见，他们是：Chris Chandler、Jim Larson、Bryce Darling、Brian McCallister、Kevin Jackson、Deepak Vohra、Pierre-Antoine Grégoire、David Dossot、Greg Donald、Daniel Bretoi、James Hatheway、John S. Griffin、Franco Lombardo和Stuart Caborn。

我们还要对Manning的工作人员致以衷心的感谢，感谢你们的支持和耐心，特别是Tiffany Taylor、Katie Tennant和Cynthia Kane，你们从不言弃。

最后最为重要的是：Martin要感谢他的妻子Veronica在他马拉松式的写作过程中所展现出的耐心；Richard也要感谢他的妻子Elisabet，即便写作占用了丈夫所有的夜晚和周末，她还是给予了坚定的支持；Eric要感谢Rossana能倾听他的抱怨，还一直提醒他写作本身就是一个不断改进、不断完善的过程。

关 于 本 书

本书主要讲解如何开发真实、稳定、版本组织合理且可维护的软件。其中理论讲得不多，更侧重于实战。我们几个人（指本书三位作者）都具有多年的系统开发经验，本书就是以实际的应用软件开发为目标，对这些经验进行提炼的结晶。Erlang编程语言本身并非我们的重点——市面上更适合用作语言教程的书有的是，本书讲述的是生产环境下的Erlang开发实践。

本书的目标是提升独立程序员或公司中的程序员团队的开发效率，因此其内容不仅涵盖了Erlang语言本身，更从头讲解了Erlang/OTP。Erlang自身具备构建强大应用的潜能，但只有借助OTP才能将这种潜能发挥出来。OTP既是一个框架，又是一组库，更是一套构建应用的方法学；它本质上是对语言的扩展。要正经学习Erlang，就一定要学习Erlang/OTP。

本书通过精心挑选的实例阐明了如何在实践中运用Erlang/OTP。通过亲手实现这些实例，你将了解如何构建稳固、版本组织合理的产品级代码，并以此榨干机架上32核机器的每个时钟周期！

本书结构

本书分为三部分。第一部分的目的是带你过一遍纯Erlang编程，并介绍一些OTP的基础知识。

- 第1章介绍了Erlang/OTP平台及其主要特性，例如进程、消息传递、链接、分布式以及运行时系统。
- 第2章是Erlang编程语言的一个简单教程，是每个专业Erlang程序员都应了解的内容的参考和总结。
- 第3章将带你开发一个通过TCP套接字通信的Erlang服务器，借此介绍OTP行为模式[①]的概念。
- 第4章介绍了OTP应用和监督树，展示了如何给第3章开发的服务器配上监督进程，并用EDoc生成文档，再一并打包成应用。
- 第5章展示了用于检测Erlang运行时系统的主要GUI工具：应用监视器、进程管理器、调试器和表查看器。

---

① Erlang/OTP中的“行为模式”（behaviour）相当于一类通用的模式框架，用户通过添加自定义回调便能够方便地实现相应的模式，详情请参见3.1.2节。——译者注

第二部分讨论实际开发相关的问题，我们将面对一些实际的开发任务，并逐渐向代码中添加各种更高级的OTP特性。

- 第6章将带你启动本书的主项目：一套用于提升Web服务器访问速度的缓存系统。本章展示一个更为复杂的多进程应用，还介绍了将监督进程用作进程工厂的方法。
- 第7章阐述了Erlang/OTP的日志和事件处理机制，并介绍了如何通过自定义事件处理程序的方式为缓存系统添加日志功能。
- 第8章介绍了分布式Erlang/OTP，解释了什么是节点，Erlang集群如何工作，节点间如何通信，以及如何借助Erlang shell的任务控制功能在远程节点上执行操作。紧接着我们将运用所学知识来实现一个分布式资源发现应用，用以发布和查询Erlang节点集群中的可用资源信息。
- 第9章将介绍Erlang内置的Mnesia分布式数据库，并展示如何利用分布式表将我们的缓存系统扩展至集群中的多个节点上。
- 第10章讲的是如何打包发布一个或多个Erlang/OTP应用。发布应用时既可以制作独立的最小安装，也可以将其附加于现有安装。另外还会介绍发布包的部署方法。

第三部分讨论如何让代码与大环境融合，如何与其他系统和用户集成，以及如何随复杂度的增加不断优化代码。

- 第11章展示了如何在TCP的基础之上为缓存应用增加REST风格的HTTP接口。通过自定义OTP行为模式，你将从头开发自己的Web服务器。
- 第12章解释了Erlang与其他语言开发的系统进行通信的基本机制。本章展示了三种接入第三方C库的途径：普通端口、端口驱动以及NIF。
- 第13章展示了用Jinterface库集成Java代码的方法，Java程序可以借此模拟成特殊的节点从而接入Erlang集群。随后我们将利用这种方法在缓存应用的后方接入一个用于提供后端存储支持的Hadoop HBase数据库。
- 第14章讲的是Erlang/OTP系统的性能评测和优化，解释了主要代码评测工具的使用并讨论了一些有助于系统调优的实现细节。

本书特意避开了`gen_fsm`行为模式，因为在实践中它的用途非常有限。本书详尽论述了有关OTP行为模式的最关键的内容，掌握这些内容，你便可以充分理解OTP行为模式，进而轻松地通过官方文档自学`gen_fsm`。`gen_fsm`的拿手好戏是二进制协议解析；但普通的`gen_server`配合恰当的模式往往更为适用，而且也更为灵活。对于希望学习`gen_fsm`的读者，我们很抱歉，不过总的来说其余的几个主要OTP行为模式会更为称手。

源代码

有别于普通文本，列于清单内或穿插于正文中的源码都使用等宽字体，像这样`fixed-width font like this`。许多代码清单都附有注释，用于强调重要概念。部分源码附有标号，分别对应于代码清单后的解释。

本书中的源码可以从http://github.com/erlware/Erlang-and-OTP-in-Action-Source下载（也可以直接到github.com上搜索书名）。出版社的站点也提供下载：www.manning.com/ErlangandOTPinAction或ituringbook. com. cn/book/828。

作者在线

购买本书的读者可以免费访问Manning出版社的专有论坛，在此你可以对本书发表评论，询问技术问题，或是向作者和其他用户寻求帮助。访问和订阅该论坛请移步http://www.manning-sandbox.com/forum.jspa?forumID=454。注册后便可在页面上找到论坛的访问方法、在此能获得哪一类帮助，以及论坛的行为准则。

Manning承诺为公众提供读者和读者之间以及读者和作者之间的交流途径，但无法承诺作者的参与度。作者在本书论坛上发帖都是自愿（且无偿）的。我们建议读者多向作者提一些有挑战的问题，好激发他们参与的兴趣。

作者在线论坛及本书出版至今的讨论合集都可通过出版社的网站访问。

关于封面插图

本书封面插图的标题叫“阿尔特温人”（An Artvinian），这是居住在土耳其东北部阿尔特温地区的居民。插图取自一本描绘奥斯曼帝国服饰的画册，这本画册由伦敦老邦德街的William Miller于1802年1月1日出版。画册的扉页已经丢失，因此很难推断准确的创作时间。画册的目录同时使用英语和法语标识插图，另外每张图片还附有两名创作它的艺术家的名字。他们一定想不到……自己的作品竟然会在两百年后被用作某本计算机编程图书的封面。

这本画册是Manning的一位编辑从古董跳蚤市场上淘来的，那地方位于曼哈顿西26号大街的Garage。卖家是个住在土耳其安卡拉的美国人，谈这桩买卖的时候他正要收工回家。当时这位Manning编辑身上没带够钱，信用卡和支票又被婉拒。而卖家当晚就要飞回安卡拉，这么一来似乎就没指望了。那最后怎么办呢？两个人最后通过握手约定的老式君子协议解决了问题。卖家提议通过银行转账付款，于是我们的编辑在纸上记下银行信息后便掖着一包画册离开了。不用说，第二天我们就把钱转了过去。一直以来，我们都深深地感激这位曾对我们的同事抱以信任的陌生人。这真让人无比怀念那充满信任的朴素的旧时代。

奥斯曼画册上的那些画，正如出我们用在其他封面上的那些插图一样，栩栩如生地展现了两个世纪前的丰富多彩的服饰。抛开当今这个过度支配的时代，它们代表着那个年代以及其他历史时期的分隔和距离。衣着习俗从那时开始改变，此前不同地域的服饰的绚丽多姿开始逐渐消失。当今，来自不同大陆的人仅靠衣着已经很难区分开来。也许乐观地看，我们正是用文化上的多样性和视觉上的差异性才换来了多姿多彩的个人生活。亦或是一种更多样、更精彩的知识与技术生活。

我们 Manning人把两个世纪前多样的地方生活融入书籍的封面，令画卷复苏，谨以此表达我们对计算机行业的创造性、首创性以及趣味性的赞颂。

引　　言

Erlang是一门以进程为核心概念的语言。什么是进程？电脑上同时运行着的多个程序，比方说文字处理软件和Web浏览器，彼此便运行在各自的进程之中。文字处理软件若崩溃，通常不会影响浏览器——反之亦然，崩溃的浏览器也不会弄丢你正在编辑的文档。进程就像是某种在并行执行流程间起隔离保护作用的隔膜，Erlang便是完全围绕进程来构建起来的。

在Erlang中创建进程易如反掌——这就像是在Java等语言中创建对象一样简单。进程变得如此廉价，使得我们得以从不同的视角来看待系统。Erlang程序中的任何独立活动都可以被视作单独的进程。没有晦涩的事件循环，也没有线程池，所有这些烦人的实现细节统统都可以抛开。即便程序需要同时运行10 000个进程来完成某个任务，也可以轻松搞定。阅读本书时你会发现，Erlang可以极大地改变我们看待系统的方式。我们希望让你看到，系统可以以更直观（也更高效）的方式得以呈现。

Erlang还是一门*函数式编程语言*。别害怕，Erlang完全可以设计得更贴近你所熟悉的那些主流语言；即便不借助函数式编程，这里记述的各种特性也一样可以实现。但函数式编程所具有的引用透明性、高阶函数、不可变数据结构等几大特点，它们本身就很值得引入Erlang。函数式编程简洁优雅地融合并呈现了这些特点。要不是引入了函数式编程，Erlang只会变得更为复杂，也不可能那么令人愉悦。

Erlang诞生记

你头一回听说Erlang时，它多半是被定性为一门“函数式并发编程语言”，而你大概也会觉得它听起来更像是某种学院派的、不实用的玩具语言。但我们要强调的是，Erlang从一开始就致力于解决真实的大规模软件工程问题。为了把话说清楚，我们得先好好聊聊这门语言背后的历史。

与C比较

Erlang和C语言的背景颇为相似。首先，二者都源自大型电信公司，都是由某个环境相对轻松的研发部门里的几个人创造出来的。两门语言的创造者都是自在之人，但他们同时也都是试图去解决具体问题的务实的工程师。对于C来说，要解决的问题是如何在硬件资源受限（相对那时而言）的情况下，用比汇编更高级的语言来开发系统软件。对于Erlang来说，问题则在于如何让

程序员开发超大规模、高并发和强容错的软件，并彻底改善生产效率、减少软件缺陷的数量——这可真不是闹着玩儿的。

两种语言的第一批追随者都来自公司内部，这些人在各种内部项目和实际产品中使用它们，并就各种务实的细节向语言的缔造者们提出了宝贵的前期反馈。二者都花了大约十年时间才为公众所知，而在此之前，它们都出色地经受住了实战的检验。C诞生于1972年左右，流行于20世纪80年代。同样，Erlang成形于1988年，直到1998年才以开源的形式对外发布。在公司外部，两种语言先点燃的都是研究机构和大学的兴趣。当然，出于历史原因它们各有各的缺点，但一般来说我们并不在乎，因为也只有它们真正地解决了问题。

让我们把目光转向过去。

20 世纪 80 年代中期，斯德哥尔摩：那个自由散漫的英国人

Erlang诞生自一个研究项目，该项目旨在寻求一种更好的编程方式，用以开发当时电信业内那些大流量、高并发、猛龙一般永生不死的控制系统。Joe Armstrong[①]于1985年加入这个项目，来到了位于瑞典斯德哥尔摩的爱立信计算机科学实验室。

这个项目的主要任务就是用尽可能多的编程语言来实现同一类通话控制系统。涉及的语言包括Ada、CLU、Smalltalk等。最终的结果结论性并不强，虽然很明显上上之选是用函数式和逻辑语言并采取高级的声明式风格来进行开发，但当时还没有哪种语言具备合适的并发模型。

但谁知道好的并发模型是个什么样呢？那时（以及之后近二十年间），并发方面的主要研究要么集中在CSP、pi演算这样的纯抽象进程模型和并发逻辑语言上，要么就集中在信号量（semaphore）、监视器（monitor）和信号（signal）这类底层机制上。

与此同时，工程师们仍然要解决各种大规模并发容错通信系统的实际问题。当时的爱立信已经有了独门秘籍，那是一套专有的揉合了编程语言和操作系统的混合解决方案，名为PLEX，AXE电话交换机的成功就要归功于它。

让人抓狂的需求

PLEX是一门相对常规的命令式编程语言，但它为后继者立下了一系列标杆：

- 进程必须是语言的核心；
- 任何进程不得损坏其他进程的内存空间，不得遗留悬空指针；
- 由于要同时跑数万乃至数十万个进程，进程创建和任务切换的速度必须要快，单个进程的内存占用量必须非常小；
- 必须能够隔离单个进程的故障；
- 必须能够在运行时对系统进行代码升级；

① 本节标题中的“英国人”指的就是Erlang之父Joe Armstrong。说他“自由散漫”是因为当时他在爱立信所参与的研究项目本身就比较发散，自由度比较大。——译者注

- ❑ 必须能够同时检测和处理软硬件错误。

与此最相近的语言当属Parlog和Strand等并发逻辑语言，但它们对进程有着不同的定义，进程的粒度更细，对单个进程的控制力也很弱。

Erlang的诞生

一天，Joe发现Prolog基于规则的编程风格可以很好地匹配他之前为描述通话控制问题而发明的手写标记法，于是他开始编写一个Prolog的元解释器[①]。通过这种方式，他扩展了Prolog，模拟出进程切换，用以并发运行多个电话呼叫。

很快，解释器所能识别的表达式形成了一个支持进程和消息传递的小型语言；虽然是用Prolog实现的，但它更简单，而且是函数式的，也没有用上Prolog的合一[②]和回溯特性。没多久，便有人一语双关地提议将之命名为Erlang（一方面指丹麦数学家A. K. Erlang，他因在通信系统统计领域的贡献而为电信工程师们所熟知；一方面它也是Ericsson Language的简写）。

就这样，开发小型但可工作的通话控制系统的需求驱动着早期Erlang的演化。特别是消息传递原语的设计，完全是出于大型电信系统的实际需要，绝非为了迎合某个特定的并发理论。这里所说的消息传递原语，指的是异步的消息发送操作符、自动消息缓存和乱序的选择性消息接收机制（这一设计深受用于规范复杂通信系统协议的CCITT SDL标记法的影响）。

1988年，一群真正的用户进行了初期实验，开发了一个全新的电信架构，事实证明该语言极大地提升了生产效率，但当时的实现实在是太慢了。于是从1990年起，Joe、Mike Williams和Robert Virding开始实现Erlang的第一个抽象机。这个抽象机名叫JAM，是一个用C写成的堆栈机，比起最初的Prolog实现要快上70倍。

1993年，第一本Erlang书籍出版，并于1996年再版。直到这时，Erlang才终于可以算得上是一门真正的语言了。[③]

发展壮大

在后来的年月里，Erlang又累积了许多特性，如分布式、记录语法、预处理器、Lambda表达式（fun语句）、列表速构、Mnesia数据库、二进制数据类型以及比特位语法等。整个系统也被移植到了Windows、VxWorks和QNX等非UNIX平台上。

1995年，随着一个巨型C++项目的分崩离析，Erlang在爱立信内部获得了空前的发展。该项目被推倒重来，这回用的正是Erlang以及“不过60个”程序员；另外还有一个靠谱的语言支持部门——OTP团队——来给他们做后盾。最终的成果便是取得了巨大成功的AXD301系统，整个系统由一百多万行Erlang代码锻造而成。

---

① 元解释器（meta-interpreter），这里指用Prolog写的Prolog解释器。——译者注

② 合一（unification），该译法取自Wikipedia（http://zh.wikipedia.org/wiki/合一）。——译者注

③ 这段历史可以参见Joe Armstrong发表于2007年的A History of Erlang，其中记载了大量Erlang的设计理念和趣闻。——译者注

与此同时，1998年，乌普萨拉大学的一群学生正将Erlang本地代码编译作为他们的硕士论文课题，并以此成立了高性能Erlang研究组；最终，HiPE（High Performance Erlang）本地代码编译器被整合进标准Erlang/OTP发行版。此外，爱立信的一个内部项目曾试图将Erlang编译为C来提高执行效率，终因生成的代码过大而失败。然而该项目却衍生出了BEAM，一个更快的、基于寄存器的多线程代码抽象机，一举取代了老旧的JAM。

但在20世纪90年代晚期，随着Java兴起，爱立信公司高层认为不该把人力投入到自创的编程语言的开发维护中去，而应该采用"全世界通用的语言"。因此，Erlang被禁止用于新项目。最终管理层被说服将Erlang开源，以便造福爱立信以外的用户。事情发生于1998年12月，很快，为数不少的核心开发者选择离职并共同创建了一家小公司，借由Erlang和丰富的电信经验，这批人很快就赚到了第一桶金。

大获成功

渐渐地，外部用户越来越多。随着时间的流逝，爱立信的人们也开始将禁令抛到脑后，Erlang的作用大得令人难以抗拒，现存的系统也不再被要求用别的语言重写了。OTP团队继续开发和维护Erlang，爱立信也持续资助着HiPE项目，以及EDoc和Dialyzer等许多衍生应用。

在学术界，Erlang则被公认为一门成熟且有价值的函数式编程语言。自2002年起，ACM SIGPLAN开始资助一年一度的Erlang Workshop，使之成为与ICFP（国际函数式编程会议）平起平坐的年度盛会。而最令人自豪的则是Erlang的并发模型成为了许多其他编程语言争相实验和效仿的对象；不过正如许多人所见，这马后炮也不是那么容易放的哟。

2006年，当硬件产业开始承认自身已然触及单核处理器的性能瓶颈时，Erlang发布了第一个支持SMP的版本，这是爱立信的OTP团队和HiPE团队通力合作的成果。接着，在2007年，Joe的新书*Programming Erlang*（《Erlang程序设计》）面世（此时距Erlang的第一本书出版已10年有余）——一时之间，Erlang变成了全球瞩目的焦点。大大小小的公司都争相用它创建各种不可思议的应用。

本书的故事便从这里开始了。

目　　录

第三部分　集成与完善

Part 1

第一部分

Erlang 起步：OTP 基础

本书第一部分深入介绍各种基本原理。我们会先快速浏览一遍语言基础，再来学习一些 OTP 的基本部件，而这些部件将成为构建贯穿于本书后续内容中各种现实场景的基石。

第 1 章

Erlang/OTP平台

本章概要

- 理解并发和Erlang的进程模型
- Erlang的容错与分布式支持
- Erlang运行时系统的重要属性
- 什么是函数式编程，如何用Erlang进行函数式编程

既然你正读着这本书，想必知道Erlang是一门编程语言——而且还是一门很有意思的语言，但正如书名所示，我们所关注的是如何用Erlang创建真实而鲜活的系统。为了实现这一目标，我们就需要OTP框架。Erlang的任何发布版本都带有这套框架，它与Erlang紧密集成，已令人难以将之与普通Erlang标准库明确地区分开来。因此，我们常用Erlang/OTP来同时指代二者或其中之一。尽管二者间的关系如此密切，但真正明了OTP的用途与运用之道的Erlang程序员却为数不多，即便真相往往只有一步之遥。就让本书来为你带路吧。

OTP 是什么意思

OTP 最初是开放电信平台（Open Telecom Platform）的缩写，Erlang 开源前这个名字多少还有点品牌效应。如今可没人稀罕它了；现在 OTP 就是 OTP。无论 Erlang 还是 OTP 都早已不再局限于电信应用；更贴切的名字应该是“并发系统平台”。

作为编程语言，Erlang可以简化高度并行分布式容错系统的构建，并以此闻名。在跳到OTP框架相关内容之前，我们会先在第2章对该语言作一个全面的综述。不过话说回来，为什么非学OTP不可呢？或许你更乐于埋头实现自己的解决方案？且让我们来看看OTP的优点：

- 生产效率——运用OTP可在短时间内交付产品级的系统；
- 稳定性——基于OTP的代码可以更集中于逻辑，并避免重新实现那些容易出错而每个实际系统又都必备的基础功能，如进程管理、服务器、状态机等；
- 监督——这是由框架提供的一套简便的监视和控制运行时系统的机制，既有自动化方式，也有图形用户界面方式；
- 可升级——框架为处理代码升级提供了一套系统化的模式；

- 可靠的代码库——OTP的代码坚如磐石并全部经过严格的实战检验。

尽管有诸多优点，但恐怕对大部分Erlang程序员来说，OTP仍然神秘莫测，必须在艰涩的文档中摸爬滚打千锤百炼方能习得。而这正是我们所要改变的状况。据我们所知，本书是第一本专注于OTP学习的书，而我们想要告诉你的是这一过程远比你想象的要轻松。我们保证你学了肯定不会后悔。

在本书结束时，你将完整地掌握OTP框架中的概念、库与编程模式，学会如何运用OTP的组件和理念开发单个Erlang程序及整套基于Erlang的系统，并令其兼具容错、分布、并发、高效和易于监控的特点。你可能还会学到一些此前未曾留意过的有关Erlang语言、运行时系统、库和工具的细节。

本章我们所要讨论的是Erlang/OTP平台中的那些用于构建OTP本身的核心概念和特性，包括：

- 并发编程；
- 容错；
- 分布式编程；
- Erlang虚拟机和运行时系统；
- Erlang的核心函数式语言。

我们不会一上来就劈头盖脸地扔出一大把概念，此处的重点是让你了解各种具体内容背后的思想，以便更好地理解后续第2、3章中论及的具体内容。Erlang很特别，本书中的诸多内容都需要花费时间去适应。在深入技术细节之前，我们希望这一章能让你明白各种机制背后的动机。

1.1　基于进程的并发编程

Erlang为了解决并发问题——也就是让多个任务同时运行——采用了全新的设计。并发是设计该语言时的核心关注点。借助进程的概念，Erlang内置的并发支持可以彻底隔离任务，令你设计出容错的架构，并充分发挥当今多核硬件的能力。不过在继续深入之前，我们有必要把并发和进程这两个术语的确切含义解释清楚。

1.1.1　理解并发

并发就是并行吗？不完全是，至少在讨论计算机和编程时二者并不等同。

有个常用的半正式定义是这么说的：“并发，用于形容那些无须以特定顺序执行的事物。”比如分别对两副牌排序，你可以排完一副再排另一副；三头六臂的话也可以两副并行一起来。这两个任务在执行顺序上不受约束，因此，它们是并发任务。它们的完成顺序也无所谓，你可以在两个任务间交替切换直至二者全部完成；倘若有多余的手脚（或是多个帮手），也可按真正的并行方式同时进行。①

① 关于“并行”与“并发”的区别，另一个常见的说法是，“并行”形容两个或多个任务在同一时间同时发生，而“并发”形容两个或多个任务在一个时间段内交替进行，同一时间内只有一个任务在执行。而本书中的定义则将“并行”列为“并发”的一个概念子集。——译者注

这听起来好像有点怪：只有同时发生的任务才能算是并发任务吧？嗯，这个定义的关键在于它们可以同时发生，而我们则可以按自己的意愿随意调度它们。有些必须同时进行的任务相互之间根本无法独立；还有些任务相互之间虽然独立却不并发，必须按特定的顺序进行，比如做蛋卷必须先打蛋。除了这些情况其余都算并发。

Erlang的一大优势就是它帮你隐藏了任务实际执行的细节。如图1-1所示，如果有额外的CPU（或核，或超线程），Erlang会利用它们并行执行更多并发任务。如果没有，Erlang会利用现有的CPU处理能力一点一点地交替执行任务。你不必操心这些细节，Erlang程序能够自动适配不同的硬件——CPU越多它们跑得越快，前提是任务的组织方式允许它们被并发执行。

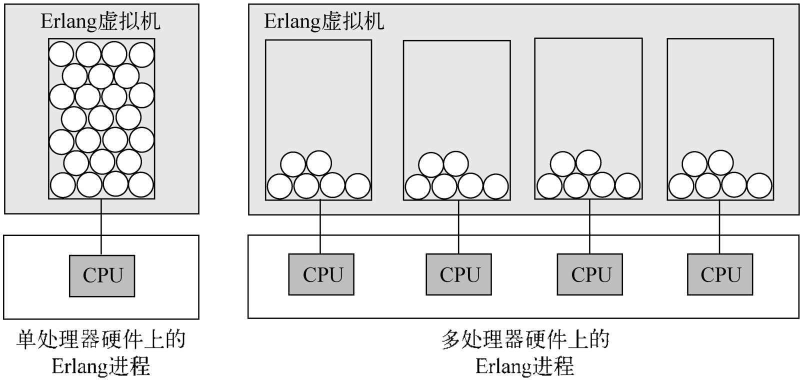


图1-1　分别运行于单处理器硬件和多处理器硬件上的Erlang进程。运行时系统会自动将负载分配到可用的CPU资源上

但若任务就是无法并发呢？若你的程序就必须先执行X，再轮到Y，最后是Z呢？这时你就得好好考虑一下待解决的问题中所隐含的实际的依赖关系了。也许X和Y无所谓谁先谁后，只要它们在Z之前完成就行。又或许X和Y各自完成了一部分的时候就可以部分启动Z。在这个问题上没有捷径可循，但往往稍微动下脑筋便收效甚佳，越有经验越容易。

重新考察问题，削减任务间不必要的依赖，可以令代码在现代硬件上运行得更为高效。但这通常不该是你的首要关注点。将程序中内聚性低的部分隔离成独立的任务，最重要的收益是更清晰可读的代码，你的精力也得以集中到实际问题上；相反，试图一蹴而就一次性完成多个任务只会令你事倍功半。这种隔离意味着生产效率的提高和缺陷数量的降低。但首先，我们需要一种更具体的表征独立任务的手段。

1.1.2　Erlang的进程模型

在Erlang中，并发的基本单位是进程。每个进程代表一个持续的活动，它是某段程序代码的执行代理，与其他按各自的节奏执行自身代码的进程一起并发运行。进程和人有些相似：个人占有的东西不能与他人共用。这不是慷慨与否的问题，以食物为例，你多吃一口，别人必定少吃一口；更关键的是，你吃了不干净的东西，也只有你一个人会生病。每个人都凭借自己的头脑和脏

器独立于他人去思考和生存。这也正是进程的行为模式，相互隔离，并确保自身内部状态的改变不对其他进程造成影响。

进程拥有自己的工作内存空间和自己的信箱，其中信箱用于存放外来消息；而许多其他语言和操作系统中的线程却是共享相同内存空间的并发活动（随之而来的是层出不穷的互踩脚趾的机会）。因此与线程相比，Erlang进程更加安全，不会有人在周围指手划脚，也不必担心有人在下一微秒出其不意地篡改自己的数据。因此我们说进程封装了状态。

进程的一个实例

以 Web 服务器为例：服务器接收网页请求，查找该网页的数据，再将数据传回请求发起方（有时会先分块，再一块一块地传输），如果请求失败则需要返回一条错误信息。显然，请求与请求之间的关联度不高[①]；然而如果服务器一次仅接受一个请求，不处理完毕就不受理下一个的话，热门站点上请求队列的长度很快就会过千。

反之，如果在请求抵达后服务器立即分配独立的进程来处理请求的话，就不再需要队列，大部分请求整体的处理时延也将趋于相等。这时单个进程所封装的状态就是请求中的URL、响应的接收方以及当前请求的处理进度。请求处理完毕后，进程退出，清理掉这个请求并回收内存。一旦某个 bug 导致某个请求失败，只有对应的那个进程会崩溃，其他进程仍完好无损。

由于进程不能直接改变其他进程的内部状态，容错便相对容易。无论一个进程执行的代码有多烂，其他进程的内部状态都不会受损。即便是在程序内较细粒度的层面上，你也同样可以设置这种隔离，就好像电脑桌面上的浏览器和文字处理器之间的关系一样。事实证明这种隔离非常有效，后续我们讲到进程监督时你就会有所体会了。

由于进程之间互不共享内部状态，它们只能进行复制式通信。一个进程要跟其他进程交换信息，就会发送一条消息。这条消息是发送方所持有数据的一个只读副本。消息传递的基本语义使分布式与Erlang自然地融为一体。现实生活中，你是无法共享线路上的数据的——你只能复制它。Erlang的进程间通信机制总会让接收方获取一份私有的消息副本，即便消息收发双方同处在一台机器上。初听起来可能很奇怪，但这意味着网络编程和单机编程完全一样！

这种分布透明性支持令Erlang程序员可以将网络视作一组资源的集合——我们不用关心进程X和进程Y是否运行在不同的机器上，因为无论它们运行在何处，通信方法都一样。我们将在下一小节对各种语言和系统中的进程通信方式做一个综述，以让你明白其中的利弊权衡。

1.1.3　4 种进程通信范式

所有并发系统的核心问题，都是信息共享，这也是所有实现者都必须解决的问题。将一个问题切分为若干不同的任务后，任务间如何通信？这个问题看起来简单，实则不然，不少大师级人

① 不过，在Web 2.0的大潮下，情况可能有所不同。——译者注

物都在这个问题上绞尽了脑汁，并经年累月地尝试了很多手段，其中一些被收编为编程语言的特性，另一些则形成了独立的库。

我们将简要讨论一下近年来受到广泛认同的四种进程通信手段。我们不打算在此耗费过多的时间，但应该足以让你大致了解当今各种语言和系统中的进程通信手段，同时我们会着重介绍它们和Erlang的区别。这4种手段分别是持锁共享内存、软件事务性内存、future和消息传递。让我们从最古老但仍旧最流行的方法开始。

1. 持锁共享内存

共享内存差不多算得上是我们这个时代的GOTO：身为当今主流的进程通信技术，它不仅历史悠久，而且跟GOTO语句一样，为你提供了一大把搬起石头砸自己的脚的办法。因为它，世代工程师都对并发产生了深深的恐惧（未曾对此心怀畏惧的人只是尚未尝试过罢了）。然而我们必须承认，正如GOTO一样，在一些底层场合中共享内存是无法取代的。

在这种范式下，两个或多个进程可以同时读写一块或多块常规内存区域。有时进程需要在这些内存区域上执行一些具备*原子性*的操作序列，其他进程在操作完成前不得访问这些区域，这就需要一种令该进程阻止其他进程访问这些区域的方法。解决之道就是锁：一种一次仅允许一个进程访问某种资源的构件。

锁的实现需要内存系统的支持，一般由硬件以特殊指令的形式提供支持。使用锁的时候进程之间必须通力合作：所有进程必须先获取锁才能访问共享内存区域，访问结束后还要将锁释放给其他进程使用。使用锁必须万分小心，差之毫厘谬以千里，因此，操作系统或编程语言分别以系统调用或语言构件的形式提供了信号量、监视器和互斥量等以基本锁为基础的高级构件，用以确保锁的请求和释放的正确性。尽管借助这些可以绕开最棘手的问题，但仍然难以克服锁的诸多缺点。随便列举几条：

- 即便冲突几率很低，锁的开销仍难以忽略；
- 它们是内存系统中的竞争热点；
- 出错的进程可能将正处于加锁状态的锁弃之不顾；
- 当锁出现问题时极难调试。

还有就是，用锁同步两三个进程还没什么问题，但随着进程数的增长，形势便会越发失控。最终很可能（在很多情况下，几乎肯定）会引发即便是经验最老到的开发者也无法预见的复杂死锁。

我们认为最好只在底层编程场合，比如在操作系统内核中，处理此类同步问题。但当今流行的大部分编程语言和脚本语言中都能看到锁的身影。它的广泛存在可能是因为锁本身的实现并不复杂，同时也不会对这些语言的编程模型造成影响。遗憾的是，虽然多处理器系统早在好几年前就已经普及，锁的广泛应用还是阻碍了我们对并发问题的思考和对大规模并发的应用。

2. 软件事务性内存（STM）

我们所要考察的第一种非传统方法就是STM（Software Transactional Memory，软件事务性内存）。目前可以在Haskell编程语言的GHC实现和基于JVM的Clojure语言中看到这种机制。STM将内存当作传统数据库，用事务来决定何时写入什么内容。通常，这种实现以一种乐观方式来规避

锁：将一组读写访问视为单个操作，若两个进程同时试图访问共享区域，则各自启动一个事务，最终只有一个事务会成功。另一个进程会得知事务失败，并应该在检查共享区域的新内容后重试。该模型直截了当，谁都不需要等待其他进程释放锁。

STM的主要缺点在于你必须重试失败的事务（当然，它们可能再三失败）。事务系统本身也会有比较显著的开销，另外在确定哪个进程成功之前，还需要额外的内存来存放你试图写入的数据。理想情况下，系统应该像支持虚拟内存那样对事务性内存提供硬件支持。

对程序员而言，STM的可控发性看起来比锁要好，只要竞争不会频繁导致事务重启，并发的优势就能充分得到发挥。我们认为该方法本质上是持锁共享内存的变体，它在操作系统层面的作用要更甚于应用编程层面。不过针对该课题的研究还很活跃，局面还可能会出现改观。

3. Future、Promise及同类机制

另一个更现代的手段是采用所谓的future或promise。这个概念还有另处一些形式；在E[①] 和MultiLisp等语言以及Java的一个库中可以找到它的身影。类似的还有Id和Glasgow Haskell中的I-var和M-var、Concurrent Prolog中的并发逻辑变量，以及Oz中的数据流变量。

其基本思路是，每个future代表一个被外包到其他进程的计算结果，该进程可能跑在别的CPU甚至是别的计算机上。Future可以像其他对象一样被四处传递，但无法在计算完成之前读取结果，必须等待计算完成。这种方法虽然概念简单、简化了并发系统中的数据传递，但也令程序在远端进程故障和网络故障面前变得脆弱：计算结果尚未就绪而连接又不幸断开时，试图访问promise的值的代码便会无所适从。[②]

4. 消息传递

正如1.1.2节所说，Erlang进程靠消息传递来通信。这意味着接收进程实际上获取了一份独立的数据副本，发送方感知不到接收方对副本所做的任何操作。向发送方回传信息的唯一途径就是反向发送另一条消息。由此而得出的一个重要结论是，无论收发双方是身处同一台机器上还是被网络所隔离，它们都能以相同的方式进行通信。

消息传递一般可分为两类：*同步方式*和*异步方式*。在同步方式下，消息抵达接收端之前发送方什么事也做不了；在异步方式下，消息一经投递发送方便可立即着手于其他事务。（在现实世界中，机器间的同步通信要求接收方给发送方回复一个确认，以告知一切OK，不过这些细节对程序员而言可以是透明的。[③]）

---

① 这里的E语言指的是一种基于JVM的并发语言，和汉语编程的“易语言”不是一个东西。——译者注

② 本段中future和promise所指的是同一个概念，因此作者在此混用。——译者注

③ 如果考虑到收发进程的故障以及数据链路的故障，那么这些细节就无法对程序员透明。消息必然会有丢失或重复的概率；一旦透明化，应用层必然要面临因消息丢失而造成的无限等待以及消息重复的风险。事实上两点之间能保证消息既不丢失也不重复的容错消息传递协议是不存在的，最多是通过时间戳、超时、重传来实现以消息重复为代价避免消息丢失的协议。在这点上即便是TCP也不例外。实践中我们能够做到的就是尽量减小丢失和重复的概率，同时减小这些错误发生时对应用造成的代价。详情可参见D. Belsnes发表于1976年的Single Message Communication以及Henry Robinson的博文：Consensus with lossy links: Establishing a TCP connection（http://the-paper-trail.org/blog/?p=110）。——译者注

同步很容易用异步实现，令接收方总是向发送方回传一个显式的回复即可。因此，Erlang中的消息传递原语是异步的。不过，发送方常常并不关心消息是否抵达——消息抵达与否其实没那么重要，因为你无法预知接收方接下来会怎样：说不定它旋即就挂掉了。这种异步的“即发即祷告”式的通信方法也意味着发送方在消息投递过程中无须挂起（特别是在慢速通信链路上发送消息时）。

当然，收发双方在这一层面的隔离并不是免费的。复制大型数据结构时成本很高，如果发送方还要保留数据副本，势必造成较高的内存消耗。在实践中，这意味着你必须在发送消息时小心掌控消息的大小和复杂度。不过一般来说，地道的Erlang程序用到的大部分消息都比较小，复制开销通常可以忽略。

我们希望以上论述能有助于你理解Erlang在当今并发编程领域中所处的位置。消息传递可能并不是其中最炫的技术，但就Erlang的发展历史来看，从系统工程角度出发，只有它最务实、最灵活。

1.1.4　用Erlang进程编程

开发Erlang程序时，你得问问自己：“在我要解决的问题中哪些活动是并发的——或者说哪些活动可以彼此相互独立地进行？”简要回答这个问题后，你便可以开始搭建系统了，你找出来的那些并发活动的每个实例都应该是系统中的一个独立的进程。

与大多数语言相反，Erlang中的并发很廉价。派生一个进程跟你在普通面向对象语言中分配一个对象的开销差不多。你可能得先好好适应一下，这个理念可真是闻所未闻！但等你适应之后，魔术便开始上演。描绘一组复杂运算，将之切分为若干并发部件，再全部建模为独立的进程。启动运算、派生进程、处理数据，在输出结果后的那一瞬间，所有进程神奇地烟消云散，它们的内部状态、它们持有的数据库句柄、它们打开的套接字，以及一切你不乐意手工清理的东西，都一并消失得无影无踪。

在这一节余下的内容中，我们将简要地看看进程是多么易于创建、多么轻量，它们之间的通信又是多么简单。

1. 创建进程：派生

Erlang进程不是操作系统线程。它们由Erlang运行时系统实现，比线程要轻量得多，运行在商用硬件上的单个Erlang系统可以轻易派生出成百上千个进程。运行时系统中所有进程之间相互隔离；单个进程的内存不与其他进程共享，也不会被其他濒死或跑疯的进程破坏。

在现代操作系统中，典型的线程会在地址空间中为自己预留数兆的栈空间（也就是说32位的机器上并发线程数最多也就几千个），栈空间溢出便会导致崩溃。另一方面，Erlang进程在启动时栈空间只需要几百字节，并且会自动按需伸缩。

Erlang创建进程的语法很直接，如下所示。让我们来派生一个执行`io:format("erlang!")`后立即退出的进程：

```
spawn(io, format, ["erlang!"])
```

这就行了。（spawn函数有若干个变体，这是其中最简单一个。）这段代码启动一个独立进程，在终端上打印文本“erlang!”后退出。

我们将在第2章给出Erlang语言及其语法的综述，但现在，在进一步阐述之前我们希望你能自行抓住示例代码的要点。Erlang的一大优点便是即便从来没见过这种语言，也能相对容易地读懂代码[①]。不妨试试看吧。

2. 进程之间怎么打交道

进程被派生并运行起来后还有别的事情要做——它们要进行信息交换。Erlang让通信变得简单。用于消息发送的基本运算符是!，读作“bang”，用法是“目的地!消息”。这就是最简单的消息传递，就像寄明信片一样。OTP框架将进程间通信提升到了另一个层面，我们将在第3章对此做深入探讨。现在，让我们来看看两个独立的并发进程间的通信，Erlang简单的通信机制一定会令你惊叹不已，详情参见代码清单1-1：

代码清单1-1 Erlang进程间通信

```
run() ->
    Pid = spawn(fun ping/0),
    Pid ! self(),
    receive
        pong -> ok
    end.

ping() ->
    receive
        From -> From ! pong        ❶ From中包含发送方ID
    end.
```

花上一两分钟看看这段代码，相信即便从未接触过Erlang你也能看懂。稍微需要注意的地方：调用了spawn的一个变体，参数是一个函数引用，该函数“名为ping、参数数目为零”；再就是函数self()，它返回当前进程的标识符，用于告知新进程该把消息回复给谁❶。

这便是Erlang进程通信的概况。每调用spawn一次都会得到一个新的进程标识符，用于唯一标识新创建的子进程。这个标识符后续可用于向子进程发送消息。每个进程都有一个信箱，无论进程繁忙与否，都会先把外来的消息存放在这儿，直到进程下次检查信箱前所有消息都寄存于此。随后进程会在自己乐意的时候用receive表达式从信箱中分检和读取消息，如同示例所示（此处是取走第一条就绪的消息）。

3. 进程的终止

进程完工后，便会消失。它的工作内存、信箱和其他资源都会被回收。该进程若是用作其他进程的数据源，那么它必须在终止前显式地将数据以消息的形式投递出去。

崩溃（异常）造成进程意外提前终止，一旦发生崩溃，其他进程会得到通知。之前我们曾说过进程之间相互独立，单个进程的崩溃不会破坏其他进程，因为它们互不共享内部状态。这构成了Erlang另一主要特性的一个支柱，该特性就是：容错。我们将在下一节做详细论述。

① 这个……真的是不敢苟同哇。——译者注

1.2　Erlang 的容错架构

在现实世界中容错就是真金白银。程序员并不完美，需求往往也不完善。正如航空工程师处理有缺陷的钢材和铝材一样，为了有效处理有缺陷的代码和数据，我们需要能够容错的系统，以防系统在遭遇突发状况时土崩瓦解。

和许多其他编程语言一样，Erlang也具备*异常处理机制*来捕获特定代码段的错误，不过它还有一套独一无二的可以有效处理进程故障的*进程链接*系统，我们即将在此进行讨论。

1.2.1　进程链接如何工作

Erlang进程意外退出时，会产生一个*退出信号*。所有与濒死进程*链接*的进程都会收到这个信号。默认情况下，接收方会一并退出并将信号传播给与它链接的其他进程，直到所有直接或间接链接在一起的进程统统退出为止（参见图1-2）。这种级联行为可以使一组进程像单个应用一样退出，因此系统整体重启时你不必担心是否还有残存下来未能完全关闭的进程。

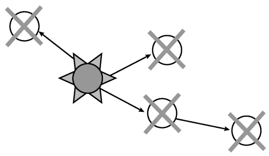


图1-2　崩溃进程发出的退出信号被传播到所有与之链接的进程，一般情况下它们会共同退出，以便完成对整个进程组的清理工作

前面我们曾提到过利用进程来清理复杂状态。其基本原理是：每个进程完整封装自己的全部状态，因此进程退出时系统的其余部分不会受损。如同单个进程一样，这一点对相互链接的进程组也同样适用。一个进程崩溃，与之协作的其他进程也一并退出，如此便可干净利落地抹掉之前建立的所有复杂状态，既节省了程序员的时间也减少了错误。

鼓励崩溃

当你还在绝望地纠结于如何挽回那些你可能根本无能为力的局面时，Erlang 的哲学却是"鼓励崩溃"——精确记录下事发位置和经过后，把一切彻底抛下重新再来。这不太常见，但的确是一条强大的容错秘诀，而且按这个思路建立起来的系统无论多复杂都可调试。

1.2.2　监督与退出信号捕捉

OTP实现容错的主要途径之一就是改写退出信号默认的传播行为。通过设置`trap_exit`进程标记，你可以令进程不再服从外来的退出信号，而是将之*捕捉*。这种情况下，进程接收到信号后，会先将其转为一条格式为`{'EXIT', Pid, Reason}`的消息，该消息描述了哪个进程出于什么原因而发生故障，然后这条消息会像普通消息一样被丢入信箱，捕捉到信号的进程就能分检并处理

这类消息了。

这类会捕捉信号的进程有时被称为*系统进程*，它们执行的代码往往有别于普通的*工作进程*（即通常不捕捉信号的进程）。身为防范退出信号进一步传播的壁垒，系统进程阻断了与之链接的其他进程和外界之间的联系，因而可用于汇报故障乃至重启故障的子系统，正如图1-3所示。我们将这类进程称为*监督者*。

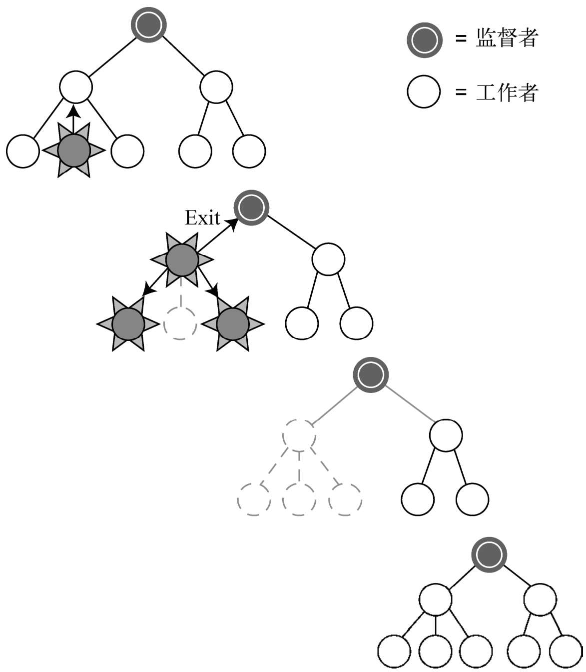


图1-3　监督者、工作者和信号：某工作进程的崩溃被级联传播至所有与之链接的其他进程，信号传播至监督者后，监督者将进程组重启。同一监督者辖区内的其他进程组则不受影响

停止并重启整个子系统的目的在于将系统恢复到一个已知的可正常工作的状态。这有点类似于重启电脑：通过重启你可以快刀斩乱麻地将电脑迅速恢复到可工作状态。但重启整台电脑的问题在于粒度太大。理想状况下，应该可以只重启系统的一部分，粒度越小越好。Erlang的进程链接与监督者共同提供了一种细粒度的"重启"机制。

不过，如果就到此为止，你还是得自己从头实现监督机制，这需要缜密的思考和丰富的经验，bug的清除和各种边界情况的处理也要花费大量的时间。幸运的是，OTP框架提供了你所需要的一切：既有运用监督机制来构建应用程序的一套方法，也有稳定的、经过实战考验的基础库。

OTP允许监督者按预设的方式和次序来启动进程。我们还可以告知监督者如何在单个进程故障时重启其他进程、一段时间内尝试重启多少次后放弃重启等。你所要做的就是提供一些参数和回调。

然而系统不应该只允许一层监督者工作者结构。在任何复杂系统中，你都可以用多层的监督树在多个层级重启子系统来解决各种意外问题。

1.2.3　进程的分层容错

通过分层可以将相关的子系统归于同一个监督者的辖区之内。更重要的是，这样做可以定义多个层级的基准工作状态，随时供你重置。在图1-4中，你可以看到两个分别受独立监督进程监督的工作进程组A和B。这两个组和它们的监督者共同形成了一个更大的进程组C，并由树中更高层的一个监督者负责。

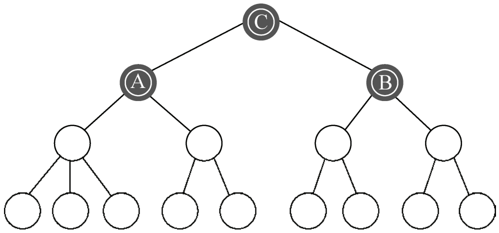


图1-4　一个分层的监督者工作者系统。如果出于某种原因监督者A崩溃或退出，它辖区内所有尚还存活的进程都会被强制关闭，同时C会收到通知，于是进程树的左半边会被重启。监督者B则不受影响，除非C决定关闭整个系统

我们假设A组进程的任务是输出供B组使用的数据流。无须B组，A组也可正常工作。更具体一点，比方说A组在处理和编码多媒体数据，B组则予以展现。我们再假设A组处理的数据中有一小部分受损，且数据损坏的模式无法在开发应用时预测。

这种畸形数据会导致A组的进程工作异常。按照鼓励崩溃的哲学，进程不会尝试去解决问题而是直接崩溃；由于进程相互隔离，其他进程并不会受到错误输入的影响。监督者检测到进程崩溃后，会将A组重启以回退到预设的基准状态，从而使整个系统恢复到一个已知的基准点。美妙的是身为展现系统的B组完全不知晓也不关心这个过程。只要A组能为B组持续提供足够的优质数据，使后者能为用户展现质量过关的内容，你的系统就是成功的。

通过隔离系统中不相关的部分并将它们组织成监督树，你可以划分出多个子系统，每个都可独立地在几分之一秒内完成重启，这样一来，即便你的系统碰上不可预期的错误也可以稳健地运行。若A组无法正常重启，它的监督者最终会放弃重启并将问题上报至C组的监督者。在这种情况下，C组的监督者会一并关闭B组然后停工。想象一下若系统中同时运行着几百个C这样的子系统，这就相当于因数据错误而丢弃了一个多媒体连接，其他连接仍然照常工作。

然而，既然大家都跑在同一台机器上，就不得不共用一些东西：内存、硬盘驱动器、网络连接，乃至处理器和所有相关电路，还有一样最重要的，就是从同一个插座上接出来的那根电源线。如果这些东西中有一样发生故障或断开，不管怎么分层怎么做进程隔离都无法避免宕机。这就把我们带入了下一个主题，也就是分布式——能助你实现最高级别的容错并令你的解决方案伸缩自如的，正是Erlang的这个特性。

1.3　分布式 Erlang

借助于语言属性和基于复制的进程通信，Erlang程序天然就可以分布到多台计算机上。要问为什么，且让我们来看两个用Java或C++这类语言写成的进程，它们运作良好并以共享内存为通信手段。假设你已经搞定了锁的问题，一切精准而高效，但就在你试图将其中一个线程挪到另一台机器上时，问题出现了。或许是为了利用更高效的计算能力和内存，或许是为了预防两个线程在硬件故障造成的宕机中同时挂掉，无论如何，这一刻降临时，程序员往往被迫重新设计代码结构，以便配合新的分布式环境中迥异的通信机制。显然，这将耗费大量的开发成本，而且很可能会引入数年才能彻底清除的bug。

Erlang程序却不受这些问题的影响。正如我们在1.1.2节所解释的那样，Erlang规避了数据共享并通过复制进行通信，这使得Erlang代码可以直接分布到多台机器上。在命令式语言里用线程编程时，各部分代码往往会因数据共享引入复杂的依赖关系；这类问题在Erlang中则很少见。今天能跑在你的笔记本上，明天就能跑在集群上。

Erlang应用通常可以直接分布到多个网络节点上，这同时意味着伸缩性问题也简化为一个数量级。你仍然需要考虑好各类进程的职能，每类进程需要运行多少个实例，在哪些机器上运行，怎样均衡负载以及怎样管理数据；但至少以下这类问题不用再劳你费心了："我到底该怎么切分现有的程序才能搭建出冗余的分布式系统？"，"它们之间该怎么通信？"，还有"我该怎样得体地处理故障？"。

实际案例

在一个雇主那里，我们在网络上跑着多个各式各样的 Erlang 应用。独立 OTP 应用的类型大概有至少 15 种，它们协同完成某个共同的任务。做集成测试时，我们固然可以在 15 个虚拟机上测试这个跑着 15 个不同应用的集群，但这绝非最便捷的做法。实际上我们一行代码也没改，就在单个 Erlang 实例上启动了所有应用并完成了测试。在那个节点上，它们使用相同的通信方式、相同的语法，就跟分布在网络中的多个节点上一模一样。

这个案例所示范的概念称作位置透明性。其基本含义是当你用进程的唯一 ID 作为目标地址向进程发送消息时，你不用了解甚至不用关心那个进程所处的位置——只要接收方还"活得好好的"，Erlang 运行时系统便会替你把消息投递到它的信箱里[①]。

现在你已经大致了解Erlang能做些什么了，下面我们来讲讲位于核心的引擎，好让你明白在Erlang程序运行时究竟发生了些什么。

1.4　Erlang 运行时系统和虚拟机

那么上述这一切背后的驱动力是什么呢？标准Erlang实现的核心是一个称作Erlang运行时系

① 作者在此处显然不够严谨，如果收发双方身处不同机器，要投递成功至少还需要双方之间网络畅通。——译者注

统（ERTS）的应用：这是一大块用C语言写成的代码，负责Erlang中所有底层的玩意儿。通过它你才能跟文件系统和终端打交道，它还处理内存，实现Erlang进程的也是它。ERTS知道如何将这些进程分布到现有的CPU资源上才能充分发挥计算机硬件的能力。同时，哪怕你只有一个单核CPU它也能实现Erlang进程的并发执行。ERTS还负责处理进程间的消息传递，并使处在不同机器上运行在各自的ERTS中的进程能够像身处同一台机器上一样进行通信。Erlang中所有需要底层支持的东西都由ERTS处理，所以ERTS移植到哪个平台Erlang就能在哪个平台上跑。

ERTS中特别重要的一个部分就是Erlang的虚拟机模拟器：这是执行Erlang程序经编译后产出的字节码的地方。这个虚拟机也就是Bogdan Erlang抽象机（BEAM）①，它非常高效：虽然我们也可以将Erlang程序编译为本地机器码，但一般没有那个必要，因为BEAM模拟器已经够快的了。注意虚拟机和ERTS之间并没有明确的界线；通常人们（包括我们自己）口中的Erlang VM指的就是模拟器加上运行时系统。

运行时系统中有许多有趣的特性，若不在文档中挖地三尺或是长期浸淫于Erlang邮件列表，你是不会知道的。它们正是Erlang能同时处理那么多进程的精要之所在，也是Erlang如此特别的原因之一。Erlang语言的基本哲学加上实现者所采取的务实方案，共同为我们带来了异常高效、面向生产的稳定系统。在这一节，我们将讨论促成了Erlang的强大和高效的3个重要方面：

- 调度器——处理运行中的Erlang进程，令所有就绪的进程共享可用的CPU资源，并在新消息到达或发生超时的时候唤醒相应的睡眠中的进程；
- I/O模型——防止系统在进程与外部设备通信时阻塞，令系统平稳运行；
- 垃圾回收器——回收不再使用的内存。

我们从调度器开讲。

1.4.1　调度器

经过多年的演进，ERTS的进程调度器提供了其他平台无法比拟的灵活性。它最初的设计目标是在单CPU上并发运行轻量级Erlang进程，而不用关心底层用的是什么操作系统。ERTS运行的时候通常就是单个操作系统进程（在操作系统的进程列表中一般名为beam或werl）。这个进程中，就跑着管理所有Erlang进程的调度器。

随着线程在大多数操作系统中的普及，ERTS也有所变化，开始将I/O系统这类东西从运行Erlang进程的线程中拿出来，放到独立的线程中去，但完成主体工作的线程仍然只有一个。如果你用的是多核系统，就必须在同一台机器上运行多个ERTS实例。不过从2006年5月起，Erlang/OTP第11版中增加了对称多处理器（SMP）支持。这是一项重大突破，令Erlang运行时系统可以在内部使用不止一个进程调度器，每个占用一个独立的操作系统线程。其效果参见图1-1。

这意味着现在Erlang进程可以以$n : m$的方式映射到操作系统线程。每个调度器处理一个进程池。可并行运行的Erlang进程最多能有m个（每个调度器线程执行一个），但同一池内的进程仍像

① Bogdan指的是BEAM的发明人Bogumil Hausman。BEAM原本叫作Turbo Erlang，后来由于复杂的法律问题更名为BEAM。详情参见4.7节。——译者注

之前所有进程共用一个调度器那样分时运行。在此基础之上，进程可以在进程池之间迁移以便维持可用调度器上的负载均衡。在最新的Erlang/OTP发布版中，甚至可以根据机器上CPU的拓扑情况将进程绑定到特定的调度器上，从而更好地利用硬件的缓存架构。这意味着，大多数时候，作为一名Erlang程序员你不用担心手头有多少CPU或有多少个核：你只要中规中矩地写程序，并尽量将程序切分为尺寸适中的并行任务就好，负载均衡之类的事情就让Erlang运行时系统去操心吧。不管是单核还是128核——都一样，只会更快。

Erlang新手常犯的一个错误就是过分依赖时序，这往往导致那些在他们的笔记本或工作站上运行良好的程序一迁移到多核服务器上就出错，因为在多核服务器上时序的不确定性要大得多。要发现这类问题，只能借助测试。好在现在的笔记本基本上都至少是双核的了，这类错误也可以被尽早发现。

Erlang的调度器还涉及运行时系统的另一个重要特性：I/O子系统。这正是我们的下一个主题。

1.4.2 I/O与调度

很多并发语言都有的一个毛病就是它们没怎么拿I/O当回事儿。单个进程进行I/O时，它们几乎都存在整个系统或大半系统阻塞的问题。这真是既恼人又没有必要，尤其是Erlang早在二十年前就已经解决了这个问题。在前一节，我们曾讨论过Erlang的进程调度器。除了处理进程调度，调度器还替系统优雅地处理了I/O问题。在系统的最底层，Erlang以事件驱动的方式处理所有I/O，当数据进出系统时，程序可以以非阻塞方式完成数据处理。这降低了连接建立和断开的频次，还避免了OS层面上的加锁开销和上下文切换。

这是一种高效的I/O处理方法。可惜，程序员往往难以分析和理解这种技术，这也是为什么只有在明确要求高可靠性和低延迟的系统中才能见到这种技术。早在2001年，Dan Kegel就在他的论文The C10K Problem中描述过这个问题，虽然现在已经略显过时，但这篇文章仍然很值得一读。它针对这个问题及可能的解决方案给出了良好的综述。这些方案实现起来全都既复杂又痛苦，这正是Erlang运行时系统替你包办这些问题的原因。Erlang在进程调度器中整合了基于事件的I/O系统。事实上，你一点儿都不用操心就能享受一切便利。这让用Erlang/OTP构建高可靠性系统变得轻松了很多。

我们要讲解的最后一个ERTS特性就是内存管理。它对进程所起的作用超出你的想象。

1.4.3 进程隔离与垃圾回收器

如你所想，Erlang跟Java等大部分现代语言一样，会自动管理内存。这儿没有显式的释放操作。相反，*垃圾回收器*会定期搜寻和回收不再使用的内存。垃圾回收（GC）算法是一片广袤而复杂的研究领域，我们无法在这儿给出详尽的阐述；不过针对那些对此有一定了解又心怀好奇的读者，可以告诉你Erlang当前使用的是一个简单明了的分代复制式垃圾回收器。

虽然实现相对简单，Erlang程序却不太会像其他语言开发的系统那样在GC时遭受停顿。这主要因为Erlang进程之间的隔离：每个进程所使用的内存都是自己的，随进程的创建和结束而分配

和释放。听起来好像没什么要紧，实则不然。首先，这意味着垃圾回收器可以在不影响其他进程运行的前提下单独暂停目标进程。其次，单个进程占用的内存通常较小，遍历可以快速完成。（也有内存占用量大的进程，但这些进程一般不用做出快速响应。）再次，调度器知道每个进程最后一次运行的时间，如果某个进程自上次垃圾回收后什么也没干，调度器会跳过它。正是这些因素让Erlang既可以轻松使用垃圾回收器，又可以保证较短的停顿时间。除此以外，有时候进程自派生到完工，再到退出，根本就没有触发过垃圾回收。这种情况下，进程的作用相当于一块昙花一现的内存，除自动分配和释放外，没有任何额外的开销。

本节所描述的运行时系统的特性使Erlang程序能够充分利用可用的CPU来运行大量进程、执行I/O操作，并自动回收内存，与此同时还能维持软实时响应能力。了解了平台这些方面的知识，便可更好地理解自己的系统自启动后的各种行为。

最后，在这一章结束前，我们还要说一说Erlang中的函数式编程。不会太罗嗦，因为我们还会在下一章详细探讨Erlang语言。

1.5　函数式编程：Erlang 的处世之道

对本书的许多读者而言，*函数式编程*可能还是个新概念。对另一些人来说则不是。函数式编程绝对算不上Erlang的决定性特征——并发才是——但它仍是该语言的一个重要方面。近年来越来越多的人意识到，函数式编程及其背后的观念天然适用于并发与分布式编程问题。（单提一下Google MapReduce就够了吧？）

要对函数式编程做一个总结的话，其主要思想就是将函数看作和整数、字符串一样的数据；运用函数调用而非`while`或`for`这样的循环结构来表达算法；以及不修改变量和值（参见附录B中对引用透明性和列表的讨论）。这些看似人为设置的约束，从工程角度来看却具有非凡的意义，Erlang程序本身也因此极为自然而可读。

Erlang并不是一门“纯粹”的函数式语言——它仍仰仗副作用。但仅限于一个操作：复制式消息传递[①]。每条消息都会对外界产生影响，同时外界也通过向进程发消息来影响它们。但每个进程本身运行的基本上都是纯函数式的程序。遵循这个模型的程序比C++和Java这类传统语言写成的程序更易于分析，同时又不至于像Haskell那样迫使你在程序中使用monad。

在下一章，我们会介绍Erlang编程语言中的关键部分。估计不少读者会觉得这个语法很别扭——它主要借鉴自Prolog而非C。虽然异于常规，它却并不复杂。忍上一段时间，它就会成为你的第二天性。待熟悉之后，你便能够看明白大部分Erlang内核模块的代码了，这才是真刀真枪的语法测试：看看你最后能否看得懂？

1.6　小结

本章我们介绍了Erlang/OTP平台中身为OTP基石的最关键的概念和特性：用进程和消息传递

① 这点也是不准确的，Erlang中的进程派生、针对进程字典的操作以及各种I/O操作都依赖于副作用。——译者注

进行并发编程、通过链接来容错、分布式编程、Erlang运行时系统和虚拟机，还有Erlang核心的函数式语言。这一切共同构成了一个稳固、高效、灵活的平台，让你得以构筑可靠、低延迟、高度可用的系统。

如果你曾有过Erlang相关的经验，这些内容大部分都不新鲜，但希望我们的讲解还算有意思，至少能让你看到一些先前未曾留意的方面。目前为止，我们还没怎么讨论OTP框架。我们把这部分内容放在第3章，不过届时进度会比较快，所以趁现在好好回味一下这些背景材料吧。接下来，第2章会针对Erlang编程语言做一个完整的综述。

第 2 章

Erlang语言精要

2

本章概要

- 与Erlang shell交互
- 数据类型、模块、函数与代码编译
- 单赋值变量与模式匹配
- Erlang语言生存指南
- 如何运用递归来编程

在上一章，我们讨论了Erlang和OTP的底层平台，却没太关注Erlang语言。Erlang本身不是本书的重点，但在你开始运用Erlang/OTP的设计模式编程之前，我们还是需要先介绍一下语言基础，以确保所有读者都能站到一条起跑线上。本章还可用作参考手册，供你通读本书时查阅。

这一章很长，学过Erlang的读者可以快速带过，不过我们也为这类读者精心挑选了一些有用的内容。这里罗列的都是我们觉得每个Erlang程序员都应了解的内容，何况即便是经验最老到的老手往往也会忽略一些重要的细节。

如果这是你第一次接触Erlang，那么你可以先只阅读本章的一部分内容，等以后需要时再回过头来查阅。我们希望这些材料足以助你消化本书后续的内容。不过在将Erlang用到正式项目上之前，你最好还是备上一本更完整的Erlang编程指南。本章的末尾列出了一些可供深入阅读的材料。在这儿我们没法教授你通用的编程技术和窍门，只能为你解释该语言各组成部分的工作原理。不过我们安排了一个针对Erlang shell使用的速成课程，来教你如何编译和运行自己的程序，并助你掌握递归。

为了更好地掌握本章的内容，你应该在电脑上安装一套可运行的Erlang。如果你用的操作系统是Windows，请在浏览器中访问www.erlang.org/download.html，从Windows Binary一栏最上方下载并运行最新版本的安装包。关于在其他操作系统上安装Erlang的步骤，参见附录A。

本章中，我们会先过一遍Erlang的基本数据类型，接着讨论模块、函数以及代码的编译运行，再就是变量和模式匹配。之后，我们将讨论函数子句与保护式（guard）、case跳转、fun函数（Lambda表达式）、异常、列表速构，以及二进制串和位串。最后，我们会讨论记录、涉及预处理器的文件包含和宏、进程操作和消息传递，还有ETS表。末了，以一段完整的针对递归编程的讨论来结束全篇。

好啦，在这一切之前，让我们先从Erlang启动后的起点开始：shell。

2.1 Erlang shell

相较于日常惯用的系统，Erlang系统是一套更富交互性的环境。使用大部分编程语言时，要么把程序编译成OS可执行文件后运行，要么用解释器来执行一堆脚本文件或编译后的字节码文件。无论哪种情况，都是让程序一路跑到结束或崩溃为止，然后回到操作系统环境中，再重复这个过程（一般是改完代码后）。

Erlang却不是这样，它更像是在操作系统中运行着的另一个操作系统。虽然Erlang的启动速度很快，但它并非被设计用于需要频繁启停的场合——它被设计用于持续运行，是为交互式开发、调试和升级而设计的。理想情况下，只有碰到硬件故障、操作系统升级之类的情况才有必要重启Erlang。

与Erlang系统的交互主要是在shell中进行的。shell就是你的指挥中心。这儿是你实验代码片段运行效果的实验室；这儿是你做增量开发和交互式调试的工地；这儿也可以是你操控线上系统的战场。为了能让你随心所欲地驾驭shell，我们给出了一些让你边看边实验用的示例。赶紧把shell启动起来吧！

2.1.1 启动shell

我们假设你已经下载并安装了Erlang/OTP。如果你用的是Linux、Mac OS X，或其他类UNIX系统，启动一个终端并运行erl命令即可。如果你在用Windows，你应该点击安装程序替你生成的Erlang图标，随后会启动名为werl的程序，它会打开一个特殊的Erlang终端，这样做可以避免直接在Windows终端下交互式运行erl会碰到的一些问题。

启动Erlang后你应该会看到如下信息：

```
Erlang (BEAM) emulator version 5.6.5 [smp:2] [async-threads:0]

Eshell V5.6.5  (abort with ^G)
1>
```

`1>`是提示符。随着你不断地输入命令，它还会依次变为`2>`等。你可以用上、下方向键或Ctrl-P/Ctrl-N键上下切换之前输入的行。另外还有几个Emacs风格的快捷键，但大部分按键的行为都比较常规。

我们还可以用`-noshell`标志启动Erlang系统，像这样（在你的操作系统命令行里）：

```
erl -noshell
```

在这种情况下，你无法通过终端与启动后的Erlang系统进行交互。要执行批处理任务或要将Erlang作为守护进程运行时可以采用这个方法。

现在你已经知道怎么启动shell了，我们再来看看你能用它来干些什么。

2.1.2　输入表达式

首先，你在shell提示符下输入的并不是什么命令，而是表达式，两者的区别在于表达式一定会返回一个求值结果。表达式求值完毕后，shell就会打印出求值结果。shell会记住这些结果，后续可以用`v(1)`、`v(2)`这样的语法来引用它们。比如，输入数值42，紧跟一个英文句号（`.`），再回车，你将看到：

```
Eshell V5.6.5  (abort with ^G)
1> 42.
42
2>
```

敲下回车后，Erlang会对表达式42求值，并打印求值结果（数值42），最后给出一个新的提示符，编号为2。不过为什么要在42后面加上一个句号呢？

1. 以句号结束

在敲下回车之前，必须用句点告诉shell表达式已输入完毕。如果不输入句号就回车，shell会一直提示你输入更多字符（提示符编号不会增加），就像这样：

```
2> 12
2> + 5
2> .
17
3>
```

要是一开始忘了句号，不用担心，补上之后敲回车就行了。可以看到，这个简单的算术表达式的求值结果仍与预期相符。现在，我们来试着取回先前的求值结果：

```
3> v(1).
42
4> v(2).
17
5> v(2) + v(3).
59
6>
```

不错，不过先别得意，我们再给你展示一个几乎所有初学者都会碰壁的问题：输入字符串，再原样输出。

2. 输入带引号的字符串

当你输入双引号或单引号字符串时（现在先不讨论二者的区别），有一个特别值得注意的问题，如果忘了结尾的引号就敲了回车，shell会把同样的提示符再打印一遍并继续等待更多输入，这跟上一个忘记句号的例子差不多。如果碰到这个情况，可以输入匹配的引号并跟上一个句号，再敲回车。例如，像这样：

```
1> "hello there.
1>
```

这里的句号并不是这个字符串结束的标志——它是字符串的一部分。为了让shell从这个状态中恢复过来，你需要加些内容来结束这个字符串：

```
1> ".
"hello there.\n"
2>
```

注意最终的字符串中包含一个句号和一个换行符，而这多半不是你想要的结果。你可以用上方向键或Ctrl-P找回那行并重新编辑，在正确的位置插入遗漏的引号：

```
2> "hello there".
"hello there"
3> v(2).
"hello there"
4>
```

默认情况下shell会保存最近20条求值结果，无论是数值、字符串还是其他类型的数据。下面，我们来仔细看看v(...)系列函数及其同类。

2.1.3 shell函数

在Erlang中有一类像v(N)这样的函数，它们只存在于shell中。这些shell函数的名字通常很短（也比较晦涩）。要获取完整的shell函数清单，可以输入help()（其本身也是一个shell函数）。初学者会难以理解这个清单，所以表2-1从头列出了你应该知道的shell函数。

表2-1 重要的Erlang shell函数

函 数	说 明
help()	打印可用的shell函数
h()	打印先前输入过的命令
v(N)	取出第N号提示符对应的计算结果
cd(Dir)	更改当前目录（Dir应是双引号字符串）
ls()和ls(Dir)	打印目录内容
pwd()	打印工作目录（当前目录）
q()	退出（init:stop()的简写）
i()	打印当前系统的运行时信息
memory()	打印内存使用信息

请立刻试一试，比如列举或更改当前目录、打印历史记录、打印系统和内存信息等。瞥一眼i()的执行结果，你可以看到正如操作系统一样，你眼前的提示符背后还跑着一大堆的东西。

现在你已经知道如何启动Erlang系统以及如何跟shell交互了，我们再来看看各种退出shell回到操作系统中去的方法。

2.1.4 退出shell

退出shell（并停止整个Erlang系统）的方法有好几种。这些方法你都应该熟悉，在管理和调试系统时它们各有各的作用。我们从对系统最友好的方法开始。

1. 调用`q()`或`init:stop()`

最安全的方法就是运行上一节提到过的shell函数`q()`。这是`init:stop()`函数的一个简写形式（你也可以直接调用这个函数），该函数以一种可控的方式关闭整个Erlang系统，它会通知正在运行的应用停止运行并给它们预留出响应时间。通常这个过程几秒内就可以完成，但线上系统可能需要花费更多的时间来完成各种清理工作。

2. BREAK菜单

如果急着退出而运行的东西也不重要，在类UNIX系统中你可以按Ctrl-C唤出底层的BREAK菜单，Windows下可以在werl终端下用Ctrl-Break唤出该菜单。它看起来是这样的：

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

其中我们感兴趣的选项是`(a)`退出系统（硬停机）、`(c)`返回shell，和`(v)`打印当前运行的Erlang的版本。其他选项则会打印出许多系统相关的原始信息，等你成为Erlang专家后，会发现这些信息对调试很有用；`(k)`还可以让你浏览所有Erlang内部活动乃至强制关闭任何故障进程，前提是你明确知道自己在做什么。注意shell本身感知不到BREAK菜单，因此当你用`(c)`返回shell时，还得再按一次回车提示符才会刷新。

3. Ctrl-G

第三个同时也是最有用的一个退出方法就是用Ctrl-G唤出用户开关命令菜单。这么做会令Erlang输出这么一段晦涩的文字：

```
User switch command
 -->
```

键入`h`或`?`并回车，你将看到以下的清单：

```
c [nn]            - connect to job
i [nn]            - interrupt job
k [nn]            - kill job
j                 - list all jobs
s [shell]         - start local shell
r [node [shell]]  - start remote shell
q                 - quit erlang
? | h             - this message
```

在提示符`-->`下键入`c`可以返回shell。键入`q`会硬停机，就跟BREAK菜单下的`(a)`一样——别把这个`q`跟shell函数`q()`弄混！后者对系统更友好。另外注意BREAK菜单位居更底层，你可以在Ctrl-G菜单中唤出BREAK菜单，反之则不行。

其余选项涉及任务控制，我们将在下一节简要介绍。

2.1.5　任务控制基础

假设你正坐在Erlang shell提示符前，一不小心写了些乱七八糟的东西，跑起来怎么也刹不住（或者你等不及它跑完）。我们时不时总会干出这种事来。这时你固然可以用上述几种方法之一去关闭Erlang系统，再重启；但更谨慎、更Erlang化的做法是取消当前任务后重启一个新任务（尤其是系统中正运行着一些不能中断的重要进程时），从而不对系统其他方面造成任何影响。

模拟一下这个情形，在Erlang提示符下输入如下内容，加上句号并回车：

```
timer:sleep(infinity)
```

（我们觉得不用对这段多作解释。）现在shell被锁死了。要解决这个问题，你应该先用Ctrl-G唤出上一节中介绍的用户开关命令菜单，然后键入j列出当前的任务。这时应该只有一个任务，所以你会看到：

```
User switch command
 --> j
   1* {shell,start,[init]}
 -->
```

键入s（在本地系统上）启动一个新的shell任务，跟之前用的那个一样，然后再看看任务列表：

```
--> s
--> j
  1  {shell,start,[init]}
  2* {shell,start,[]}
-->
```

要连接到新任务上，你可以明确输入c 2。另外由于2号任务已经被*标记为默认选项，也可只键入c：

```
--> c
Eshell V5.7.2  (abort with ^G)
1>
```

你又回来了！但是等等，原来的任务哪儿去了呢？再键入Ctrl-G，列出任务，你会看到它仍旧挂在那里。我们可以输入k 1来关掉它，然后再返回shell继续试错：

```
User switch command
 --> j
   1  {shell,start,[init]}
   2* {shell,start,[]}
 --> k 1
 --> j
   2* {shell,start,[]}
 --> c
```

干这类事情的时候，务必确认你关闭的是哪个任务，尤其是在手头同时进行着多个不同的任务的时候。一个进程被关闭后，它所有的历史记录、先前的求值结果，以及一切与这个shell任务相关联的东西都会消失。等我们在第8章讨论分布式Erlang和远程shell时你还会看到Ctrl-G菜单的更多用法。它简单而强大，是远程控制和在线系统调试的利器。

现在你应该已经找到了使用Erlang控制台的感觉，该来看看Erlang语言了。

2.2 Erlang 的数据类型

学习任何编程语言都必须了解如何表示基本数据。Erlang的内置数据类型简单明了，为数也不多，但你可以用它们干很多事。我们将按以下顺序讲解这些数据类型：

- 数值（整数和浮点数）；

- 二进制串/位串；
- 原子；
- 元组；
- 列表（和字符串）；
- 唯一标识符（pid、端口、引用）；
- Fun函数。

Erlang中的数据通常被称作项式（term）。阅读这一节时你可以借助示例边试边学。（回车前别忘了句号）。让我们先从最简单的讲起。

2.2.1　数值与算术运算

Erlang有两种数值类型：整型和浮点型（float）。大部分算术运算都会自动进行类型转换，因此通常用不上显式强制类型转换（详情参见下一节）。

1. 整数

Erlang中的整数大小没有限制[①]。较小的整数会被存放在单个机器字长内；处理较大的整数（即所谓的bignum）时，会自动按需分配内存。这些对程序员而言都是完全透明的，所以你不必担心截断或溢出的问题——这些情况压根儿就不会发生。

一般来说，整数的写法没什么特别的（不妨试着输入一些大整数看看）：

```
101
-101
1234567890 * 9876543210 * 9999999999
```

另外，你还能使用从2进制到36进制的整数（分别采用数字0～9加上字符A～Z/a～z），当然2进制、16进制以及8进制以外的进制并不常见。这种写法借鉴自Ada编程语言。

```
16#FFffFFff
2#10101
36#ZZ
```

还有，利用下面的$前缀记法可以得到任意字符的数值编码（ASCII/Latin-1/Unicode皆可，试试看）：

```
$9
$z
$\n
```

等我们在2.2.6节讨论字符串时你还会再看到这个记法。

2. 浮点数

浮点数采用64位IEEE 754-1985格式（双精度），其语法与大部分编程语言相同，唯一的区别是许多语言允许浮点数以小数点开头，如.01，而Erlang要求必须以数字开头，如0.01：

```
3.14
-0.123
299792458.0
6.022137e23
6.6720e-11
```

① 唯一的限制就是物理内存的大小。——译者注

Erlang没有单精度浮点数。C/C++/Java中的float指代单精度浮点数，有这些语言背景的程序员需要特别注意这一点。

3. 算术运算与位运算

Erlang中常用的算术运算符都采用常见的中缀表示法，`+`、`-`、`*`的行为也符合常规。如果参与二元运算的两个参数中有浮点数，运算将被转为浮点型，Erlang将在必要时自动将整型参数转成浮点型。例如，`2 * 3.14`的结果是浮点数6.28。

除法则分两种。首先是`/`运算符，它总是返回浮点数：例如，`4/2`的结果是2.0，而不是2。整除运算（运算结果会被截断）则由`div`运算符实现，如`7 div 2`，结果是3。

整除运算对应的余数可由运算符`rem`实现，如`15 rem 4`，结果是3。（涉及负数时结果取决于取模运算。）

其余的浮点函数位于标准库模块math中，直接根据C标准库中对应的函数命名，如`math:sqrt(2)`。

另外还有一些用于整数位运算的运算符：`N bsl K`表示将整数*N*左移*K*位，`bsr`是右移。位逻辑运算符分别是`band`、`bor`、`bxor`和`bnot`。例如，`X band (bnot Y)`会在*X*中将*Y*所占用的位清零。

关于数值数据类型就介绍到这里，我们再来看看另一类同样基础的数据类型：位和字节。

2.2.2 二进制串与位串

*二进制串*就是无符号8位字节的序列，用于存放和处理数据块（通常是读自文件或通过某网络协议接收到的数据）。*位串*则是广义的二进制串，其长度不必是8的整数倍，如一个半字节共12位。

长度不受限制的位串是最近才加入语言的新特性，按字节对齐的二进制串则已经有多年历史了。但对程序员而言二者的表面差异并不大，只是如今可以借助位串完成一些以往无法完成的轻巧任务。由于语法相同，*二进制串*的概念又如此深入人心，现在已经没什么人（包括我们）再用位串这个说法了，除非是有必要强调长度方面的灵活性。

二进制串的基本语法如下：

```
<<0, 1, 2, ..., 255>>
```

也就是一个包含在`<<...>>`内的逗号分隔的整数序列，整数取值范围为0～255。两边的分隔符不能含有空格，如不能写成`< <`。二进制串内可以容纳任意多个字节，例如，`<<>>`表示一个空二进制串。

我们还可以用字符串构造二进制串，比如：

```
<<"hello", 32, "dude">>
```

这与直接输入字符串中相应字符的8位编码（ASCII/Latin-1）的效果相同。所以，这种写法仅限于8位字符，不过在处理基于文本的协议时常常会用到它。

这些简单示例只展示了如何创建长度为8的整数倍的二进制串。Erlang还有一种更高级也更复

杂的语法，用于构造新型的二进制串或者说位串，并可针对位串进行模式匹配和数据提取。稍后我们将在2.10节再给出一些示例。

下一个主题要讨论的数据类型，对Erlang程序员来说几乎跟数值和二进制位同样基本，它就是：原子（atom）。

2.2.3　原子

在Erlang中，原子是一种仅由字符序列来标识的特殊字符串常量，因此两个原子只要具有相同的字符表示，就完全等同。但在系统内部，这些字符串存放在某张表内，并由表的下标定位，因此在运行时只要比较两个小整数就可以判断两个原子是否相等。每个原子也仅占一个字长的内存。（特定原子的下标是在运行时自动分配的，但系统每次运行的时候分配的下标可能不同；用户无从知晓，也无须知晓这一点。）

在Erlang中，原子的作用类似于Java或C中的`enum`常量：用作标签。区别在于无须事先声明原子；你可以随意创建并随处使用各种新的原子。（不妨在shell中试试后续的示例。）在Lisp编程语言中，它们被称为符号（symbol）。比起数值常量，用原子编程更简单、更可读，对用户也更友好。

通常情况下，原子以小写字母开头，如：

```
ok
error
foo
undefined
trap_exit
```

在首字母之后，可以使用大写字母、数字、下划线和@，如：

```
route66
atoms_often_contain_underscore
pleaseDoNotUseCamelCaseInAtomsItLooksAwful
vader@deathstar
```

如果还要用到其他字符，你就得给它们加上单引号（当然你也可以给前面的那些原子加上单引号——有时为了表述明确，确实会在文档之类的地方这么做）：

```
'$%#*!'
'Blanks and Capitals can be quoted'
'Anything inside single-quotes\n is an atom'
```

你应该把原子当作一类特殊的标签，而不是普通的字符串。它们的长度上限是255个字符，在单个系统中原子的总数也有一个上限：目前是一百多万（准确地说是1 048 576）。一般来说这个上限已经足够大了，但对于长期运行（数天、数月、数年）的系统，你应该避免动态生成诸如`'x_4711'`、`'x_4712'`这类全局唯一的原子。原子一经创建，即便不再使用也永远不会被清除，除非系统重启。

以下几个原子几乎所有Erlang程序都会用到：

- `true`和`false`用于布尔运算；

- ok用于那些返回值没有任何实际意义、仅通过副作用发挥作用的函数（在C或Java中，这个场景可以用void来解决）；
- undefined用作表示未知量的占位符。

现在我们已经学完了基本数据类型，该来看看如何创建更为复杂的数据结构了，让我们从元组讲起。

2.2.4 元组

元组（或*n*元组，即三元组、四元组等的一般形式）是其他Erlang项式的定长有序序列。元组用大括号来表示，如：

```
{1, 2, 3}
{one, two, three, four}
{from, "Russia", "with love"}
{complex, {nested, "structure", {here}}}
{}
```

可以看到，元组可以包含零个（{}）、一个（{here}）或多个元素。这些元素可以是同一类型，也可以是不同的数据类型；这些元素本身也可以是元组或任意其他数据类型。

Erlang的一个标准约定是用原子作为第一个元素来标记元组数据的类别，如{size, 42}或{position, 5, 2}。这称为*标记元组*（tagged tuple）。

正如C中的struct或Java中的对象一样，元组是在Erlang中构造复合数据结构或一次性返回多个结果值的主要手段；只是元组的元素项没有名称，只有编号（从1到*N*）。访问元组中的元素是常数时间复杂度的操作，跟在Java中访问数组元素一样快速（和安全）。借助后续即将介绍的*记录语法*，你可以给元组中的元素项命名，这样就不用直接使用下标了。另外，通过*模式匹配*还可以简单地用变量来引用元组的不同部分，因此很少需要直接用下标来访问元素项。

标准库中的模块实现了一些更为复杂的*抽象数据类型*，如数组、集合、字典（即关联数组或哈希表）等；但在底层，它们大都是采用各种手段基于元组实现的。

元组针对的是定长序列。要处理变长序列，你就需要用到列表。

2.2.5 列表

列表是Erlang各种数据类型中真正的主力军——在这点上大部分函数式语言都是相通的。原因在于列表的简单、高效和灵活，还有就是它天然遵循*引用透明性*——其基本思想是被名称所引用的值不可更改（详情参见附录B）。列表可用于存放任意多个项式。我们用方括号表示列表，以下是一些列表最简单的例子：

```
[]
[1, 2, 3]
[one, two, three]
[[1,2,3],[4,5,6]]
[{tomorrow, "buy cheese"},
 {soon, "fix trap door"},
 {later, "repair moon rocket"}]
```

所以，列表就是零个或多个Erlang项式（项式本身也可以是列表）的序列。空表[]也被称为nil，这个名字源自Lisp编程语言，它更像是个原子，特别是它的值也只占一个字长的内存。

添加列表元素

有件事用元组无法简单高效地完成，用列表却可以，那就是以现有的列表为基础创建一个新的、更长的列表，并使原列表成为新列表的一部分。|（管道符）的作用便在此。例如：

```
[ 1 | [] ]
```

它将|右侧的空表与左侧的元素1合并，得到列表[1]。请在shell中试试，亲眼看看这个示例是如何工作的。以此类推：

```
[ 2 | [1] ]
```

可得到列表[2,1]（注意顺序：新元素应从左侧添加）。你也可以一次性添加多个元素，但只能从|左侧添加：

```
[ 5, 4, 3 | [2,1] ]
```

这将得到[5,4,3,2,1]。此处的添加顺序同之前添加的1和2一样，先添加3，再是4，最后是5，只是任务分解工作由编译器代为完成了而已。

另外，你还可以用++运算符向列表追加任意长度的列表。例如：

```
[1,2,3,4] ++ [5,6,7,8]
```

可以得到列表[1,2,3,4,5,6,7,8]。其过程还是一样：先是[4|[5,6,7,8]]，再是[3|[4,5,6,7,8]]，以此类推，最后是[1|[2,3,4,5,6,7,8]]。++右侧的列表不会被修改——Erlang不允许这类破坏性修改——它只是借由一个指针成为了新列表的一部分。这也意味着++运算符不关心右侧列表的长度，因为根本就用不上。

左侧的列表可就不一样了。要想以前面介绍的方式构造新列表，就必须先找到左侧列表的末尾（这个例子中是元素4），再从后往前逐步完成新列表的构造。也就是说*左侧列表的长度决定了++运算符的耗时*。有鉴于此，新内容（通常较短）应该尽量从左侧加入列表，即便最终得到的列表是逆序的也无妨。随着新元素的加入，列表越来越长，与其每次都为了在列表末尾添加元素而反复遍历列表，还不如最后用一个函数调用快速反转逆序的列表（在这点上请务必相信我们）。

Erlang还用列表来表示另一种常用数据：文本字符串。

2.2.6　字符串

Erlang中的双引号字符串实际上就是列表，其元素就是该字符串中各字符的数值编码所对应的整数。比如下列这些字符串：

```
"abcd"
"Hello!"
" \t\r\n"
""
```

它们与以下列表等价：

```
[97,98,99,100]
[72,101,108,108,111,33]
[32,9,13,10]
[]
```

还可写作：

```
[$a, $b, $c, $d]
[$H, $e, $l, $l, $o, $!]
[$\ , $\t, $\r, $\n]
[]
```

（你还记得前几页讨论整数的那一节中提到的`$`语法吧）。这层对应关系在Erlang标准库的某些函数的名字上也有所体现，如`atom_to_list(A)`，它返回任意原子`A`中所有字符组成的列表[①]。

字符串就是列表，也就是说你先前学到的所有处理列表的方法也同样适用于字符串，许多字符串编程任务仅靠基本的列表处理技巧就足以完成。然而，这种设计也有缺点，其中之一就是当你拿到一个包含若干小整数的列表时很难判断它到底是不是个字符串。

字符串与shell

Erlang shell为了区别对待字符串和普通列表，会检查列表的元素是否全部为可打印字符。如果是，就打印成双引号字符串，否则就打印为整数列表。这样对用户更为友好，但偶尔也会不如愿（例如，当某表达式返回一个看似由可打印字符组成的整数列表时，你就会看到一行乱码）。

有一个适用于这种状况的技巧，就是在列表起始处加一个零，迫使shell打印出实际内容。例如，若`v(1)`显示为字符串，`[0 | v(1)]`就不会。（当然你也可以全权掌控打印策略，用标准库函数中的字符串格式化函数来美化打印结果，但那多没劲儿啊？）

现在你已经知道如何构造复杂数据结构了，我们马上讲解剩下的基本数据类型：标识符和fun函数。

2.2.7 pid、端口和引用

这3种标识符数据类型密切相关，因此我们一并在此进行介绍。

1. pid（进程标识符）

你已经知道，Erlang支持用进程编程，任何代码都需要一个Erlang进程作为载体才能执行。每个进程都有一个唯一标识符，通常称作pid。pid是一种特殊的Erlang数据类型，应被视为一种不透明对象。但shell会以`<0.35.0>`这样的格式打印pid——即包含在尖括号内的3个整数。在shell中你不能用这个语法创建pid；该格式仅用于调试目的，以方便你对pid进行比较。

尽管可以认为在系统生存期内所有的pid都是唯一的（除非重启Erlang），但事实上当系统的运行时间长到一定程度，前前后后创建的进程总数过亿时，进程标识符就有可能被重用。不过一般来说这不是什么问题。

`self()`函数能告诉你当前进程（即调用`self()`的那个进程）的pid。请在shell里试一试——没错，shell本身也是一个Erlang进程。

① 即返回字面上和该原子相同的字符串，如将原子`'abc'`转为字符串`"abc"`。——译者注

2. 端口标识符

*端口*与进程差不多，只是还能与Erlang外界通信（其能力基本上也就仅此而已——尤其是，端口不具备代码执行能力）。因此，*端口标识符*与pid密切相关，shell打印端口的格式为`#Port<0.472>`。本书后续还会针对端口进行讨论。

3. 引用

这类数据类型中的第三种数据类型就是*引用*（常被称作ref），可由`make_ref()`函数生成（试一下！），其shell输出格式为`#Ref<0.0.0.39>`。引用常被用作各种要求保证唯一性的一次性标签或cookie。

2.2.8　将函数视作数据：fun函数

Erlang被称为*函数式*语言，这类语言的一个显著特征就是可以像处理数据一样处理函数——也就是说，函数可以成为别的函数的输入，也可以成为别的函数的求值结果，还可以把函数存在数据结构中供后续使用，诸如此类。当然，还要提供函数调用机制。在Erlang中，这种将函数包装成数据的对象称作fun函数（也称作*Lambda表达式*或*闭包*）。

我们将在2.5节大略介绍函数，然后在2.7节详细阐述fun函数。请注意虽然shell按`#Fun<...>`的格式打印函数（尖括号内是一些调试信息），你却无法用这个语法来创建fun对象。

至此，最后一种内置数据类型也介绍完了。下一个话题需要综合所有内置数据类型：比较运算符。

2.2.9　项式的比较

Erlang的各种数据类型有一个共同点：它们都可通过内置的`<`、`>`和`==`运算符进行比较和排序。常规的数值排序自然不在话下，比如`1 < 2`以及`3.14 > 3`等，原子、字符串（以及其他各种列表）和元组则按字典序排序，因此有`'abacus' < 'abba'`、`"zzz" > "zzy"`、`[1,2,3] > [1,2,2,1]`以及`{fred,baker,42} < {fred,cook,18}`。

这些都还算常规；但除此之外，不同类型之间也有一套排序规则，比如`42 < 'aardvark'`、`[1,2,3] > {1,2,3}`以及`'abc' < "abc"`。更准确地说，数值小于原子，元组小于列表，而原子既小于元组也小于列表（别忘了字符串也是列表）。

你没必要强记不同数据类型间的排序规则。只要记住*任意*两个项式都可比较，且结果总是确定的就可以了。尤其是在用标准库函数`list:sort(...)`对列表进行排序时，即便列表中混有多种类型的项式（数值、字符串、原子、元组……），也总能得到一个排好序的列表，其中最前面的是数值，接着是原子，以此类推。你可以在shell中试试：比如`list:sort([b,3,a,"z",1,c,"x",2.5,"y"])`。

1. 小于或等于/大于或等于

在语法上，Erlang跟大多数语言（Prolog除外）有一个细微的区别，就是小于或等于运算符*不*写作`<=`，原因在于这看起来太像是个指向左侧的箭头（这个符号也确实被用作左箭头）。取而

代之，小于或等于写作`=<`。大于或等于运算符则跟大部分语言一样，写作`>=`。总之，你只要记住比较运算符看起来绝不像箭头就行了。

2. 相等比较

Erlang有两种相等比较运算符。第一种是*完全相等*，写作`=:=`，仅当运算符两侧完全等同[1]时才返回`true`。例如，`42 =:= 42`。其否定形式（*不完全相等*）写作`=/=`，如`1 =/= 2`。

一般来说，判断两个项式是否相等时更倾向于采用完全相等运算符（我们后续将要讨论的模式匹配也用它来比较项式）。但这会导致看似相等的整数与浮点数被判为不相等。比如，`2 =:= 2.0`的结果就是`false`。

按数学法则对数值（或包含数值的元组，如向量）进行比较时，一般应该改用*算数相等*运算符，写作`==`。必要时它会将整数强制转换为浮点数再进行比较。这样一来，`2 == 2.0`返回的就是`true`了。其否定形式（*算数不等*）写作`/=`。例如，`2 /= 2.0`返回的是`false`。但请记住，针对浮点数做相等判断总是有风险的，浮点数的机器表示法伴有微小的舍入误差，这可能会在本应相等的数值间引入些微的偏差，使`==`返回`false`。涉及浮点数时最好只用`<`、`>`、`=<`或`>=`进行比较。这些都是算术运算符——必要时它们都会将整数转为浮点数。这就是为什么之前你能用`>`来比较3和3.14。

`==`（支持数值类型转换的算术相等比较运算符）的正确性是得不到保障的——除了算术运算以外，它基本上总有问题——在程序中使用`==`只会难为Dialyzer这类帮你定位程序不良行为的程序分析工具。这么做还会掩盖一些本该早早现出原形的运行时错误，以至于只有在文件或数据库中出现莫名其妙的数据时你才会意识到问题的存在（比如显示成1970.0的年份或是显示成2.0的月份）。

所以，老练的Erlang程序员是不用相等比较运算符的，他们会尽量使用2.4.3节所讨论的模式匹配。

2.2.10 解读列表

与大多数常见编程语言中的列表不同，Erlang的列表非常与众不同，在这个有关数据类型的话题结束之前我们有必要给予它一些特别关注。

1. 列表的结构

列表一般是由空表（nil）和所谓的*列表单元*共同构成的。这些单元各自携带一个元素挨个儿挂接到现有列表的顶部，从而在内存中形成一个单链表。每个单元仅占用两个字长的内存空间：一个用于存放元素值（或指向元素值的指针），称为*首部*（head），另一个是指向列表其余部分的指针，称为*尾部*（tail），参见图2-1。有时，有Lisp或函数式编程背景的人会将列表单元称作cons单元（源自list constructor），装配cons单元的动作也被一些喜欢标新立异的人称作consing 。

① 完全等同的意思是值和类型都必须相同。——译者注

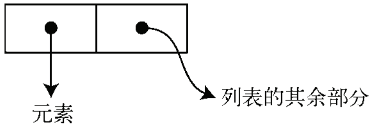


图2-1　列表单元——列表的基本构件（两块相邻的单字长内存区域）

尽管从技术角度看列表单元的首尾元素没有什么差别，但习惯上第一个（首部）总是用于保存该单元所携带的元素值，第二个（尾部）则指向列表的其余部分。列表的语法以及所有针对列表操作的库函数都遵循这个惯例。

初学者经常踩到的一个坑就是误把管道符写成逗号（老手们也时常会犯这个手误）。看看这个例子：以下两个表达式有什么区别?

```
[ 1, 2, [3,4] ]
[ 1, 2 | [3,4] ]
```

（看看你能否随着阅读的逐步深入看出些端倪。想要提示的话就到shell里去试试吧。）

答案揭晓，第一个表达式是一个包含3个元素的列表，其中最后一个元素本身也是一个列表（包含两个元素）。第二个表达式是一个在列表`[3,4]`上堆叠两个元素后形成的包含四个元素的列表。图2-2展示了这两个表达式的结构。在继续学习前请务必把这部分内容理解透彻——这是有关列表的一切知识的核心。另外可参考附录B中对列表和引用透明性所做的更为深入的论述。

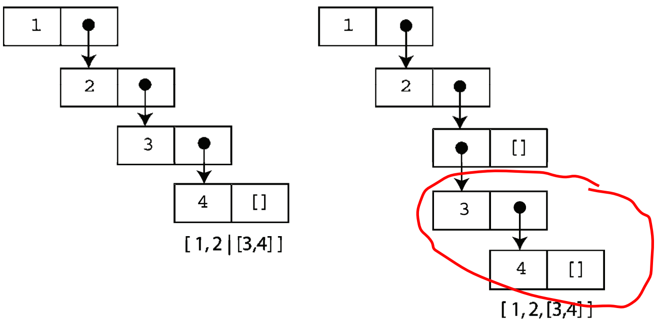


图2-2　正文示例中的列表单元结构。左侧是将2和1堆到`[3,4]`顶部而成的一个简单列表。右侧的列表则包含3个元素`[1,2,x]`，其中最后一个元素x是列表`[3,4]`

你会慢慢地爱上列表。但请记住，列表主要用于存放临时数据（比如用作当前正在处理的数据的容器）、编排中间结果，或是用作字符串缓冲。对于需要长期存储的数据，如果尺寸因素很关键，你可能就得考虑用别的办法了，比如用二进制串来存放较大的字符串常量数据。

随后你会看到，Erlang中包括字符串操作在内的大部分数据处理，都可归结于列表遍历，跟你在大多数其他语言中遍历容器和数组差不多。列表将是你的主要中介数据结构。

2. 非严格列表

最后一个要注意的问题就是严格列表与非严格列表的区别。迄今为止，你碰到的都是严格列表。它们在最内层都以一个空表作为尾部。也就是说，从最外层开始一次摘掉一个单元，最后剩

下的那个单元的尾部肯定是个空表。

非严格列表则是在非列表数据之上堆叠列表单元而成的列表。例如：

```
[ 1 | oops ]
```

这样就构成了一个尾部不是列表的列表单元（此处以原子'oops'作为尾部）。Erlang并不禁止这样做，也不会在运行时对这种情况进行检查。但一般来说，要是看到这样的东西，那多半是程序的某些地方写错了。

非严格列表的主要问题在于，很多函数要求输入参数必须是严格列表，如果传入一个非严格列表，这些函数在遍历列表时最终都会看到一个不是列表的尾部，进而导致崩溃（抛出异常）。即便你觉得自己想到了一个很高明的点子，也不要冒险以这种方式去使用列表单元——这样做很容易出错，既会迷惑人也会迷惑程序分析工具。虽然的确存在那么一两种正确运用非严格列表的方式，但它们已经是超出本书讨论范畴的高级编程技巧了。

有关Erlang基本数据结构的讨论就到此为止了，我们即将进入下一个主题，看看程序的创建以及如何编译执行程序。首先，我们来聊聊代码的居所：函数以及模块。

2.3 模块和函数

目前为止，你看到的都只是基本的Erlang表达式：可经由计算而返回某些值的代码片段。实际的Erlang程序可没法在shell里一行搞定。为了赋予代码活络的结构和稳定的居所，Erlang将模块用作代码的容器。每个模块的名字都是一个全局唯一的原子。Erlang标准库中包含大量预定义模块，比如内含诸多列表处理函数的lists模块。

本节我们先解释一些Erlang函数相关的细节：如何调用函数、函数元数的重要性、标准库，以及什么是BIF。然后我们将展示如何创建自己的模块、如何编译，以及如何运行模块中的函数。我们还会简单解释一下已编译模块和在shell中输入的代码之间的区别。首先，我们来看看如何调用模块中的函数。

2.3.1 调用其他模块中的函数（远程调用）

要调用其他模块中的函数，需要在函数名之前以冒号为分隔符加上函数所处模块的名字。例如，要调用标准库模块lists中的reverse函数来反转列表[1,2,3]，应写成（你可以用shell试试）：

```
lists:reverse([1,2,3])
```

这种形式的函数调用被称为*远程调用*（调用其他模块中的函数），与此相对应的是*本地调用*（调用同一模块中的函数）。不要将此处的远程调用与远程过程调用（remote procedure call）混淆，那是分布式编程中的一个完全不同的概念（指的是让别的进程或计算机帮你执行某个函数）。

在上面的例子中，函数的参数只有一个。用Erlang编程时，需要特别注意这类细节。原因将在下一节解释。

2.3.2 不同元数的函数

函数参数的个数称作*元数*。例如，具有1个参数的函数叫*一元*函数，具有2个参数的函数叫*二元*函数，具有3个参数的函数叫*三元*函数，以此类推。另外还有你已经见过的像`self()`这样的*空元*函数——也就是没有参数的函数。

跟其他大多数编程语言相比，函数的元数在Erlang中要重要得多。Erlang中没有函数重载之类的机制，然而，即便两个函数使用同一原子作为函数名，只要它们的元数不同，Erlang就会将它们视作两个完全不同的函数。因此，函数的全名必须包含元数（以斜杠为分隔符）。比如，前面提到的列表反转函数的全名是`reverse/1`；或者，若要强调函数所在的模块，则应写作`lists:reverse/1`。不过请注意，该语法仅用于需要函数名的位置，如果将`hello/2`用作表达式，Erlang会将之解释成原子`'hello'`除以2（这可行不通）。

为了进一步说明，不妨看一下函数`lists:reverse/2`，它的作用与`reverse/1`几乎一样，但它会进一步把第二个参数追加到最终结果中去；因此，`lists:reverse([10,11,12],[9,8,7])`会返回列表`[12,11,10,9,8,7]`。在某些语言中，为了避免命名冲突，这个函数可能不得不被命名为`reverse_onto`之类的名字，但在Erlang中两个函数可以用同一个原子命名。请不要在编写你自己的函数时滥用这个能力——命名应当以系统化的方式进行，这样才容易让用户记住如何调用你的函数。如果你写的函数在命名上只有元数不同，功能却大相径庭，最终结果可能无法让你满意。为命名而摇摆不定时，请尽量选用区分度较高的名字。

无论如何，请记住在讨论具体某一个函数时一定要说明元数，只提函数名是不够的。

2.3.3 内置函数和标准库模块

和其他编程语言一样，Erlang也自带一套由各种实用函数组成的*标准库*。这些函数散布在大量模块中，各模块应用广泛程度不一。名为erlang的那个模块尤其重要，它包含了整个Erlang系统最核心的函数，一切都以它为基础。还有一个非常实用的模块就是你已经碰到过的lists模块。io模块负责基本的文本输入输出处理。dict模块提供了基于散列的关联数组（字典），array模块提供了可扩展的、带整数索引的数组，诸如此类不一而足。

有些函数涉及非常底层的内容，以致于不得不将它们集成到语言和运行时系统的内部。这些函数通常称作*内置函数*（BIF），它们和Erlang运行时系统一样，都是用C语言实现的。（有人可能会对这个定义的一些细节有异议，但这算是BIF最通俗的定义了。）需要强调的是，erlang模块中所有的函数都是BIF。某些BIF，像我们在前一节碰到的`lists:reverse/1`，原则上完全可以直接用Erlang实现（正如lists模块中的其他大多数函数那样），但为了追求执行效率还是采用了C来实现。一般来说你不必关心这些函数是怎么实现的——从外部看来它们都一样。但鉴于BIF这个术语在Erlang界如此常用，仍然有必要了解一下它所指的是什么。

erlang模块中的某些重要函数在程序和shell中的应用都很广泛。因此它们会被自动导入，也就是说不必显式给出模块名。比如你曾见过的用于返回当前进程标识的`self()`函数。这是一个针对`erlang:self()`的远程调用，但由于该函数属于自动导入函数，可以省略`erlang:`前缀。

别的例子还包括用于启动新进程的`spawn(...)`，以及用于计算列表长度的`length(...)`等。最后，Erlang语言中的运算符实际上也是隶属erlang模块的内置函数。比如，`erlang:'+'(1,2)`也可写作`1+2`。

好了，我们已经讨论了标准库中现有的模块。但如何创建自己的模块呢？谜底即将揭晓。

2.3.4 创建模块

shell中的表达式已经玩够了，要想自行开发实际的Erlang程序，就必须把编写的代码放置到一个或多个模块中。要新开发一个可供自己或他人重用的模块，你需要做几件事：

(1) 编写源码文件；

(2) 编译；

(3) 加载已编译的模块，或将它放到加载路径中以便自动加载。

第一步很简单——启动你最拿手的文本编辑器（Notepad都行），打开一个新文件，就可以开始开发了。给模块取一个名字，接着用这个名字加上.erl后缀作为文件名保存文件。以下的代码清单展示了一个名为my_module.erl的示例文件。赶快拿起你的文本编辑器创建这个文件吧。

代码清单2-1 my_module.erl

```
%% This is a simple Erlang module

-module(my_module).

-export([pie/0]).

pie() ->
    3.14.
```

让我们从最简单的部分切入，写有`pie() -> 3.14.`的这部分称作*函数定义*。它创建了一个函数`pie`，该函数不接受任何参数，并返回浮点数3.14。*函数首部*（包含函数名和参数列表）和*函数体*（描述了函数的用途）由箭头`->`隔离。请注意这里不需要return之类的关键字：*函数的返回值就是函数体中表达式的值*。同时请注意函数定义的末尾必须附上句号（`.`），这和在shell中必须在表达式末尾追加句号是一样的。

第二个值得注意的地方就是第一行的*注释*。Erlang的注释用`%`表示，待会儿我们还会讨论到注释。

除了注释，模块的第一行一定是模块声明，其格式为`-module(...).`。总的来说，Erlang中既非函数也非注释的东西都属于*声明*。声明以连字符开头（`-`），并跟函数定义一样，必须以句号结尾。模块声明是不可或缺的，且它指定的名字必须与文件名相符（即除去.erl后缀以外的部分）。

最后要说的是`-export([...]).`这一行。这是*导出声明*，它会告知编译器哪些函数是外部可见的。此处没有列出的函数都是模块的内部函数（所以也就无法在shell中调用它们）。这个例子只有一个函数，想要用它，就得把它放入导出列表。前一节曾解释过，只有同时给定函数名和元数（此处是0）才能唯一确定一个函数，因此，这里要写成`pie/0`。

继续之前，我们先来解释一些和注释相关的问题。

注释

Erlang源码的注释格式只有一种。注释从%开始到行尾结束。当然，字符串和原子中的%不算。例如：

```
% This is a comment and it ends here.

"This % does not begin a comment"

'nor does this: %'    %<-but this one does
```

如果乐意，你也可以在shell中写注释，不过这没什么意义，所以前面我们也就没有讲到这点。

根据规范，与代码同处一行的注释以一个%开头，独占一行的注释以两个%开头，如：

```
%% This is your average standalone comment line.
%% Also, longer comments may require more lines.

frotz() -> blah.   % this is a comment on a line of code
```

（有些人甚至喜欢用3个%开头的注释行来描述整个文件层面的信息，譬如位居源码文件顶部的注释。）

遵循这些规范的理由之一是Emacs和erlIDE等具有语法感知能力的编辑器能够辨别这些注释，并会根据行首%的数量自动缩进注释。

现在你已经有了一个源码文件，并在其中定义了一个模块，该进行编译了。

2.3.5　模块的编译和加载

编译模块时，会产生一个和模块对应的扩展名为.beam而非.erl的文件，其中包含可被Erlang系统加载执行的指令。与源码相比，它更为精简和高效，同时包含了系统加载并运行模块所需的一切内容。相比之下，源码文件还有可能通过包含声明来引用其他文件（参见2.12.2节）。模块被编译时，构成该模块完整源码的所有文件都会被读取。由此，单一的.beam文件才是模块更为确定的形态，尽管其可读性很差，也无法手工编辑——你只能先修改源码再重新编译。

1. 在shell中编译

要想在做实验和测试时编译模块，最简单的办法就是调用shell函数`c(...)`，它不光负责模块的编译还能完成模块加载（前提是编译通过），以便你即刻进行测试。该函数以Erlang shell的当前目录为相对路径来寻找源码文件，你甚至可以省略模块名末尾的.erl。例如，你若在之前建立的文件所在的目录下启动Erlang，便可以这么做：

```
1> c(my_module).
{ok,my_module}
2> my_module:pie().
3.14
3>
```

`c(...)`返回的结果`{ok,my_module}`表示编译通过，生成了一个名为my_module的模块，并完成了加载。你可以通过调用该模块导出的函数`pie`来进行验证。

检查源码文件所在的目录（可以用shell函数`ls()`完成），你会发现除my_module.erl之外现在

又多出了一个名为my_module.beam的新文件。这就是已编译版本的模块，也称作目标文件。

2. 模块加载与代码路径

退出Erlang shell（比如用shell函数`q()`）后再于同一目录下重启Erlang，试试跳过编译环节直接调用刚才的模块（假设前面已经编译通过）：

```
1> my_module:pie().
3.14
2>
```

这是为什么？很简单：当Erlang尝试调用某个尚未加载到系统中的模块时，只要能找到与模块名对应的.beam文件，它就会自动尝试加载。查找.beam文件的目录由代码路径指定，默认情况下，当前目录也包括在内（你的.beam文件也就是在这儿被找到并加载的）。

调用`code:get_path()`函数可以检查当前的代码路径设置。该函数返回一个列表，你会看到该列表的第一个元素是个句号（.），它表示当前目录。默认代码路径还包括标准库目录。此外，你还可以通过code模块中的函数随意调整这些路径。

2.3.6 独立编译器erlc

真实的软件项目一般都倾向于使用GNU Make这类外部工具将构建过程脚本化。在这种情况下，你可以用独立的erlc程序直接从操作系统命令行里启动编译器。例如：

```
erlc my_module.erl
```

（当然你可以亲手跑跑看，试一下！）这和之前用shell函数时有些不同。在这里，你需要给出包括.erl扩展名在内的完整文件名。你还可以像用C编译器那样指定各种选项，比如，这样写可以指定输出目录（放置.beam文件的位置）：

```
erlc -o ./ebin my_module.erl
```

（可能你已经注意到在`code:get_path()`的例子中包含在代码路径内的所有标准库目录都以/ebin结尾。按照Erlang的惯例，存放.beam文件的子目录应取名为ebin。回头等我们讲到应用时再做详细说明。）

Windows下的情形要复杂一些：安装程序并未将erl和erlc程序所在的目录加入`PATH`环境变量；你必须先自行添加，然后才能在cmd.exe命令行中运行这两个程序。它们位于Erlang安装目录的bin子目录下——路径可能会是C:\Program Files\erl5.7.3\bin之类。另外请记住erl与cmd.exe有点儿合不来——它更适合于运行Erlang脚本，但若要用到交互式环境，最好还是用werl（点击Erlang图标时打开的就是它）。

2.3.7 已编译模块与在shell中求值

在Erlang shell求值的表达式与放在模块中（完成编译、加载并运行）的代码是不同的。我们说过，.beam文件是模块的一种高效、可部署的形态。同一份.beam文件中的代码全部是在同一时间、同一环境下编译的。模块中的代码可以完成一些模块相关的任务，如指定导出函数，查询模

块名称，或声明模块相关的各种其他信息等。

而输入到shell中的代码本质上只是一些转瞬即逝的一次性表达式。这些代码不属于任何模块。因此，在shell中是无法使用声明的（如`-export([...]).`或`-module(...).`），这里缺乏这类声明发挥作用所需的模块语境。

shell通过动态解释的方式对表达式进行解析和求值。在执行效率上，这种做法比已编译代码要低效得多（会差上若干个数量级），但如果只是调用现存的已编译模块中的某个函数，就无所谓了（该函数本身会以正常速度执行）——例如在shell中执行`lists:reverse([1,2,3])`。这时，shell只要准备好列表`[1,2,3]`并将之传给`reverse`函数便可（当然完事儿后还要打印结果）。虽然shell的速度相对较慢，人也无法感知其间的差别。

然而，借助列表速构（参见2.9节）或巧妙地运用匿名递归函数（一种会让初学者绞尽脑汁的漂亮技巧）[①]，仍然能在shell中写出几乎完全用解释器执行的代码，并以此来完成大量工作。这种做法的执行效率显著低于放在模块内编译执行的代码。所以，请记住：不要用shell解释器来评测代码的执行效率。要想得出有意义的评测数据，就必须把代码写成模块，而不是直接丢到shell里。不要以shell的行为为依据对执行效率妄下结论。

注意　在少数边界情况下，同一段代码在shell下的行为与编译到模块内的行为会有些微的差异。遇到这种情况时，应以编译版本为黄金准则。shell在解释代码时总是竭尽所能地与编译版本的行为保持一致。

直到目前为止我们所有的示例中都缺少一样东西。发现了没？你都还没用到过变量呢！其原因在于变量与模式匹配密切相关，我们想把它们放在一起介绍。现在你已经了解了基本数据类型、模块，还有函数，该来看看怎么在代码中使用变量了。

2.4　变量与模式匹配

Erlang的变量有别于大部分其他编程语言中的变量，这也是为什么我们直到现在才介绍它。它们并不比其他语言中的变量复杂，反倒要简单很多！它甚至简单到会让你第一反应觉得：“这种东西有什么用？”

这一节中，我们先介绍变量、单次赋值（single assignment）和=运算符的工作机理，再深入介绍模式匹配，同时展示如何运用模式匹配轻松提取数据结构的各个部分或对它们做断言。

---

① 这里讨论的递归匿名函数问题实际上就是Y Combinator。Erlang shell只能接受表达式，但在Erlang中具名函数定义并不是表达式（参见2.5节），因此在Erlang shell中无法定义具名函数，只能使用fun来定义匿名函数。另一方面，在Erlang中只能通过递归来实现循环，这样就引入了如何定义匿名的递归函数的问题。解法参见http://stackoverflow.com/questions/867418/how-do-you-write-a-fun-thats-recursive-in-erlang。关于Y Combinator请参见http://en.wikipedia.org/wiki/Fixed_point_combinator。——译者注

2.4.1　变量的语法

Erlang变量最显著的特点就在于变量名必须以大写字母开头！（还记得吧？以小写字母开头的名字已经被用于原子了。）以下是一些变量的示例，变量名中的单词以驼峰体（CamelCase）隔开，这是Erlang变量的标准命名风格：

```
Z
Name
ShoeSize12
ThisIsARatherLongVariableName
```

变量名也可以以下划线开头。这种情况下，按常规第二个字符通常应该是大写字母：

```
_SomeThing
_X
_this_may_look_like_an_atom_but_is_really_a_variable
```

两种命名方式有一点小小的功能性区别：赋值之后一直未被使用的变量往往会触发编译警告。这个机制可以帮我们发现大量低级错误，所以不要关闭这个警告。然而，如果使用某个变量的目的仅在于提高程序可读性，你便可以在变量名前加上一个下划线。（等我们讲到模式匹配的时候你便会明白为什么要这样使用变量。）这样一来即便这些变量不被使用，编译器也不会报警。同时，所有未被使用的变量都会被优化掉，不会带来额外的成本：这样一来，你就可以毫无顾虑地以改善可读性为目的使用它们标注程序。

2.4.2　单次赋值

下一个出乎意料之处就是，Erlang的变量被严格地限定为只能接受*单次赋值*。也就是说，变量一旦被赋值——或者用Erlang界的话说，变量被*绑定*到某个值上，该变量在其整个*作用域*（即变量在程序中生效的范围）内便一直持有这个值。在程序的不同位置可以重复使用同一变量名，但仅限于互不覆盖的多个不同的作用域内，所指的当然也是不同的变量。（这就好像德克萨斯州的巴黎和法国的巴黎之间的区别一样）。

在大多数其他编程语言中，变量就像是起了名字的盒子，在程序中你随时都可以修改盒子里的内容。仔细想想，就会觉得奇怪，这跟你在代数课上学到的内容可不一样。Erlang的变量却与对应的数学概念相符：它就是指代某个值的一个名字，且这种指代关系不会背地里悄悄改变（否则方程可就解不出来了）。这些值就在计算机内存中的某处，但Erlang不用你事必躬亲地操心各种细节，诸如创建盒子、在盒子间搬运东西，或是重复利用盒子以便节省空间。Erlang编译器会帮你搞定一切，而且表现不俗。

要想了解更多关于单次赋值和*引用透明性*概念的内容，请参见附录B。

1. =运算符以及在shell中使用变量

Erlang中最简单的赋值就是使用=运算符。这是一个*匹配运算符*，你将会看到，它的能力可不止是简单的赋值。但当前，请先在shell中试试这个例子。

```
1> X = 42.
42
2> X.
42
3> X+1.
43
4>
```

这个结果应该跟你料想的一样。但shell中的变量还有点特殊。它们的作用域是“除非我反悔，否则只要shell还在跑，变量就生效”。你可以调用shell函数`f()`来让shell遗忘先前绑定的所有变量，就像这样：

```
4> f().
ok
5> X.
* 1: variable 'X' is unbound
6> X = 17.
17
7> X.
17
8>
```

可以看到，一旦被遗忘，`X`就可以被用作其他用途了。要是还没遗忘原值就尝试重用会怎么样呢？

```
8> X = 101.
** exception error: no match of right hand side value 101
9>
```

唔，受单次赋值的限制，你会接到一个异常。错误信息说发生了一次匹配操作[①]。如果仍用`X`的原值来赋值呢？

```
9> X = 17.
17
10>
```

这回通过了。这就是匹配的含义：如果`X`已经被绑定到某个值，匹配运算符会检查右侧的值是否与之相等（比较时使用的是完全相等运算符，忘了的话可以翻阅前文的2.2.9节）。要想在不影响其他shell变量的绑定的前提下遗忘`X`，可以使用`f(X)`，像这样：

```
10> Y = 42.
42
11> f(X).
ok
12> X.
* 1: variable 'X' is unbound
13> Y.
42.
14>
```

但请记住这只是shell中变量作用域的工作方式。在模块内，作用域依赖于函数定义之类的东西，在作用域之内是无法提前遗忘变量绑定的。等到2.5.2节我们还会深入讨论相关细节。

2. 变量及其更新

接受了无法更新变量的值的事实之后，你多半会困惑那到底该怎么进行各种修改呢？毕竟，

① 并且失败了。——译者注

程序的任务大多是利用原有数据计算新数据——例如，将某个数加1。简单地说，要想追踪其他值，就给它另起一个名字。例如，假设变量X指代某个整数，要给X加1得到的值起个名字，就可以写作X1 = X + 1：

```
1> X = 17.
17
2> X1 = X + 1.
18
3>
```

你也可以换用其他描述性更强的名字，像是NewX或IncrementedX。这种做法是优是劣，取决于手头代码的实际情况（太长的名字也会有损可读性）。要是没有起名字的灵感，也可以退而求其次，用X1、X2、X3这类名字来指代X的变型。（如果你在纳闷怎么处理循环中的变量，那还得等到2.15节我们讨论递归函数的时候才能得到解答。）

在某些场合，你会遇到大量基本相同又有些微差异的数据，从而不得不动用大量不同的变量。一般来说，应该尽量避免这类场景。这种时候应该尝试将代码切分为独立的函数，令每个函数拥有自己的X，并专心解决整个问题中的单个分解步骤。长远来看，这样可以大大改善代码的可读性和结构。

变量本身是很乏味的。不管有没有单次赋值的限制，每一种实际的语言都支持变量。变量的真正威力要通过模式匹配才能体现出来。

2.4.3　模式匹配：加强版的赋值

模式匹配是Erlang不可或缺的功能。一旦上手，你就会惊讶自己以前在没有它的情况下都是怎么熬过来的，今后只要想到要用不支持模式匹配的语言来编程就会万分沮丧。（请相信我们。）

模式匹配有如下重要作用：

- 选定控制流分支；
- 完成变量赋值（绑定）；
- 拆解数据结构（选择和提取各个组成部分）。

现在就让我们来揭穿=运算符的那些阴暗的秘密吧。

=运算符就是模式匹配

在前一节，我们将=称作匹配运算符。这是因为它的功能就是*模式匹配*，而不是赋值。运算符的左侧，是一个*模式*；右侧，是一个普通表达式。做匹配运算时，首先计算右侧的表达式，得到一个值。接着拿该值去匹配左侧的模式（有点像用字符串去匹配正则表达式）。若模式匹配不上，比如17 = 42或true = false，则匹配宣告失败并抛出一个原因代码（reason code）为badmatch的异常。在shell中，该异常体现为一条错误信息“no match of right hand value ...”。

若匹配成功，在左侧模式中出现的所有变量都会与右侧值中的相应组成部分绑定，然后程序将继续计算紧随其后的表达式。（如果整个模式仅由单个变量组成，如X = 42，则该变量会与整个右侧值绑定。）为了演示这个过程，请在shell中做如下实验：

```
1> {A, B, C} = {1970, "Richard", male}.
{1970,"Richard",male}
2> A.
1970
3> B.
"Richard"
4> C.
male
5>
```

不难看出此处发生了些什么。模式{A,B,C}与右侧的元组相匹配，于是，各个变量分别与相应的元素绑定，令你得以在后续的代码中引用它们。简单得不能再简单了。

以下是另一种常见的匹配形式：

```
1> {rectangle, Width, Height} = {rectangle, 200, 100}.
{rectangle,200,100}
2> Width.
200
3> Height.
100
4>
```

此处，模式要求元组的第一个元素必须是原子rectangle（用作标签）。由于右侧元组中的第一个元素正是与之相匹配的原子，元组的元素数目也恰好为3个，匹配成功，于是变量Width和Height被绑定。

同一变量在同一模式中可以出现多次，在要求两个字段的值必须相等时可以用上这个技巧：

```
1> {point, X, X} = {point, 2, 2}.
{point,2,2}
2> X.
2
3>
```

如果对应的字段不相等，匹配就会失败：

```
1> {point, X, X} = {point, 1, 2}.
** exception error: no match of right hand side value {1,2}
2>
```

由于单次赋值的限制，不可能将1和2都赋值给X，无论先后。

2.4.4　解读模式

模式与表达式形态类似但能力却有限。其中只能包含变量、常量以及常量构成的列表和元组等数据结构，运算符、函数调用、fun函数等都不行。但在此范围内模式可以任意复杂、嵌套任意多层。举个例子，我们先来创建一个列表，其中包含某系统中的用户信息（当前仅有一个用户）：

```
1> Users = [{person, [{name,"Martin","Logan"}, {shoe_size,12},
     {tags,[jujitsu,beer,erlang]}]}].
...
2>
```

（Shell会回显你刚输入的值，简洁起见，我们省略了回显的内容。）现在，我们来抽取列表中第一个用户的某些特定信息：

```
2> [ {person, [{name,_,Surname},_,{tags, Tags}]} | _ ] = Users.
...
3> Surname.
"Logan"
4> Tags.
[jujitsu,beer,erlang]
5>
```


首先，注意这里用到了单下划线（_）表示的省略模式（don't-care pattern）。也就是说，在模式的某处用上_的话就表示你不关心右侧相应位置上的值。同一模式中可以多次出现下划线，如同这个例子一样，但不要求右侧相应位置上的值都相等（普通变量则有这个要求）。省略模式有时也被称作匿名变量，但它们根本不是变量，只是占位符。

其次，看一下该模式的最外层，尤其是=左侧最后，形如[... | _]的那部分。要想解读这个模式，请回忆一下我们在2.2.10节对列表的讨论：列表由列表单元构成，呈链状，单个列表单元可写作[...|...]。你可以把这些列表单元画出来，可以看到它们就像洋葱一样一层覆着一层，最中心是一个空表，其余每层各自携带一些数据。

于是，上面那个模式可以解读如下：

> 它的最外层是列表单元，其内层数据（模式中的| _]部分）我不关心，但其携带的数据（模式中的[... |部分）应具备{person, [...,...,...]}这样的结构——即一个带person标记的二元组，其第二个元素是一个含有3个元素的列表。这3个元素中的第一个是带name标记的三元组，我需要知道它的第三个元素的值——我们将其命名为Surname；第三个元素是一个带tags标记的二元组，我需要知道它的第二个元素的值——我们将其命名为Tags。

看明白了吗？那可真是恭喜了。不过这正体现了模式的威力，不到50个字符的一行表达式便表达了如此长篇累牍的业务逻辑（用其他语言来写的话非得洋洋洒洒写上一大段不可）。这便是模式的特点：自然、紧凑、易读、强大。

用++完成字符串前缀匹配

你应该还记得，Erlang的字符串（双引号内的字符串）就是字符编码的列表。这就简化了字符串的前缀匹配。先来看一个简单列表的前缀匹配的例子：

```
[1,2,3 | Rest] = [1,2,3,4,5,6,7]
```

由于左右两侧的前3个元素相同，匹配得以成功。于是，变量Rest被绑定到3之后的列表单元：也就是，Rest = [4,5,6,7]。

然而字符串是字符编码的列表，同时你又可以用$语法得到字符的编码值（例如，$A会得到65），所以以下代码也凑效：

```
[$h, $t, $t, $p, $: | Rest] = "http://www.erlang.org"
```

这会将Rest绑定到字符串"//www.erlang.org"上。效果不错，但还可以更简洁些。前面我们说过模式中不可出现运算符。只有一个例外：用于拼接字符串的++运算符。当且仅当左侧参数是字符串常量时，++可以在模式中出现。于是上面的例子就可以写作：

```
"http://" ++ Rest = "http://www.erlang.org"
```

（我们把两个斜杠也放到了模式中，这样Rest就变成了"www.erlang.org"，有意思吧？）整个过程一点儿也不稀奇。针对"abc" ++ SomeString，++运算符实际上构造了一个[$a, $b, $c | SomeString]形式的列表，与你前面手写的模式一模一样。当然，为了让编译器正确地展开表达式，++运算符左侧的参数必须是一个编译期字符串常量。

由于基本字符串匹配的简化，正则表达式在Erlang中并不多见。用正则表达式实现这种程度的匹配未免太小题大作，并且还会带来额外的开销。

到这里我们该给你鼓鼓劲儿了，截至目前为止，Erlang中绝大部分怪异、艰深的内容你都已经见识了。诸如原子、元组之类的各种概念，同时担当着字符串角色的古怪的列表，变量要用大写字母开头而且只能赋值一次，函数的元数居然是函数名的一部分……好啦好啦，从今往后，就是一片坦途了。只要抵御住第一轮的文化冲击，Erlang便不再艰深。闲话少说，让我们来用函数做些真正有意思的事情吧。

2.5　函数与子句

在讲解函数之前一定要把各种基础知识都讨论清楚才行！但最关键的还是要充分理解变量和模式匹配，只有这样你才能畅通无阻地掌握Erlang的函数。倒不是说Erlang的函数在概念上与其他语言有多大差异；只是Erlang的函数与模式匹配之间的联系十分密切。尽管这种联系比较直观，但也需要耗费些精力才能适应。

我们在2.3.4节中讨论模块时，你已经见识了简单的函数定义。鉴于你已经学会如何自己编写模块，从现在起，我们不再频繁用shell来举例（不过你仍然需要用shell来编译和执行代码）。对于后续内容中的前几个示例，你可以继续使用先前建立的模块——名为my_module的那个——将要测试的新函数添加进来即可。尝试调用函数前，记得先将函数名（包括相应的元数）加入-export([...])列表并用c(my_module)重新编译模块。如果看到“undefined function my_module:xxx/N”这样的错误信息，那八成是遗漏了上述步骤中的某些环节。如果看到“undefined shell command xxx/N”，那是因为你在尝试调用函数时忘了加my_module:前缀。

2.5.1　带副作用的函数：文本打印

我们从基本任务做起：获取一些输入并将之打印到终端。在Erlang中一般使用标准库函数io:format(...)来向标准输出流写入文本。它要求两个参数：第一个参数是格式字符串，第二个是欲打印项式的列表。你马上就会在自己的函数print中用上它。你所要写的这个函数将接受一个变量Term作为参数：

```
print(Term) ->
    io:format("The value of Term is: ~p.~n", [Term]).
```

将这个函数写入你的模块，在导出列表中加入print/1，通过shell用c(my_module)再次编译模块，再调用my_module:print("hello")。你将看到如下信息：

```
1> c(my_module).
{ok,my_module}
2> my_module:print("hello").
The value of Term is: "hello".
ok
3>
```

格式字符串中的转义码~p表示以美化方式打印Erlang项式。它会将可打印字符列表显示为双引号字符串，另外如果项式过长而无法在一行内完整显示，便会分多行显示（并适当缩进）。用你在2.2节学到的各种数据类型来试试你的新print函数。用它来打印列表[65,66,67]会发生什么？

（格式字符串中的转义码~n表示“插入一个换行符”，这样消息之后便会多出一个新行，不过你多半已经猜到了吧。）

现在如果将~p换成~w（试一下！）并重新编译，然后再像之前那样用my_module:print("hello")调用该函数，你会看到：

```
5> c(my_module).
{ok,my_module}
6> my_module:print("hello").
The value of Term is: [104,101,108,108,111].
ok
7>
```

那个难看的列表是什么？嗯，还记得字符串就是字符编码的列表吧？转义码~w表示“以原始形态打印Erlang项式”，既没有花哨的换行，也不会把列表打印成双引号字符串，即便列表中的元素都是可打印字符对应的编码也一样。

io:format(...)函数是带有*副作用*的函数的一个实例。你可以看到，该函数会返回一个结果值（原子'ok'），但其主要目的是对其周遭环境产生某种作用。（以此类推，print函数也是一样的。）实际上，Erlang中的所有副作用都可被视作*消息*（而且它们往往也就是以这种方式实现的）。在这个案例中，io:format函数会先准备好要打印的字符串，再将之作为消息发送至终端驱动，最后返回'ok'。

最后，要知道你已经开始在Erlang shell中进行*交互式开发*了：修改代码、重新编译并加载新的版本进行试验，整个过程都无须停止和重启Erlang环境。如果你的Erlang系统还在后台跑着其他要紧的东西（比如向客户提供网页服务），它仍然能够在你修修补补的同时欢快稳健地运行。

2.5.2　用模式匹配在多个子句中进行选择

接下来，我们来看看模式匹配是怎么掺和进来的。Erlang的函数可以有多个*子句*。而上一节的示例中只包含一个子句，以下的示例则有3个子句：

```
either_or_both(true, _) ->
    true;
either_or_both(_, true) ->
    true;
either_or_both(false, false) ->
    false.
```

此处的either_or_both/2函数是布尔函数的一个实例——即专门处理true和false值的函数（它们本身也是Erlang的两个普通原子，还记得吗？）正如其名，它应该具有与内置的or运算符相同的功能：当任意参数或所有参数都为true时，结果为true；否则结果为false。同时该函数不接受非布尔值参数。[①]

请注意子句由分号（;）分隔且最后一个子句由句号（.）结尾。同一函数的所有子句必须具备相同的函数名和相同的参数数量，且必须在同一处定义——不允许在同一函数的两个子句之间再插入其他函数定义。

子句选择

调用函数时，Erlang会自上而下地尝试对子句进行模式匹配：首先，用输入参数去匹配第一个子句中的模式；若匹配失败，就尝试下一个子句，以此类推。就这个示例而言，只要第一个参数是true，便会选中第一个子句（第二个参数是什么则无关紧要——请注意该示例在第二个参数的位置上用了一个省略模式）。

第一个子句若匹配不上，而第二个参数为true，则会选中示例中的第二个子句。但若该子句也无法匹配，就会尝试第三个子句，若仍然匹配不上，最终会抛出一个function_clause类型的运行时异常，以示本次函数调用的参数与所有子句都不匹配。

现在，我们再来看看这些子句，并思考你在每一步都掌握着哪些信息。若步入第二个子句，你便知道第一个参数肯定不是true（否则第一个子句就能匹配）。与此类似，你若一路走到了第三个子句，你便知道前两个参数都不是true。此时，余下的唯一一种合法的组合便是两个参数都是false（前提是参数只能取值为true或false）。

在良好的编程实践中，这种信息应该显式存在于代码中——这就是为什么最后一个子句不接受（false, false）以外的任何内容。要是有人用foo或42等意料之外的值来调用该函数，便会得到一个运行时异常（function_clause），而这正是你期望达到的效果：这意味着这类错误调用会快速崩溃，以便尽早地发现错误和修正代码，这样错误数据才不会进一步蔓延到系统的其他部分。你若想在这儿充好人，在最后一个子句中加上(_, _)并对其余所有情况都返回false，那么像either_or_both(foo, bar)这样的调用就不会触发任何错误警示而直接返回false。

2.5.3　保护式

尽管如此，仍有漏网之鱼。要是有人用either_or_both(true, 42)或either_or_both(foo, true)来调用前面的函数，它只会平静地返回true，仿佛一切太平无事。你可以使用保护式来添加额外的约束，从而堵住这个漏洞：

```
either_or_both(true, B) when is_boolean(B) ->
    true;
either_or_both(A, true) when is_boolean(A) ->
    true;
either_or_both(false, false) ->
    false.
```

① 此处“不接受非布尔值参数”的说法实际上是有问题的，作者将这个问题留到了2.5.3节去解决。——译者注

子句保护式由关键字when开始到->箭头结束。其中可以包含一个或多个由逗号分隔的判定，仅当所有判定都为true时该子句才会被选中。可以看到，现在你需要在模式中引入变量，以便在保护式中对它们进行引用，所以这个示例用变量名A和B取代了省略模式。is_boolean(...)判定是内置函数之一，因此你不必指定模块名（内置函数都位于erlang模块中）。所有基本数据类型都有相似的判定函数：is_atom(...)、is_integer(...)等。is_boolean(...)判定函数用于检查参数值是否为原子true或false。

包括此类*类型判定*在内，能用在保护式中的操作是十分有限的。大部分运算符都可以用（+、-、*、/、++等），部分内置函数也可以用（比如self()），但你不能调用自定义的函数或其他模块中的函数。原因部分在于效率——子句选择必须足够快，但更主要的是这些函数可能带有副作用。重要的是，某个保护式*失败*后（被判为false），必须能够*若无其事*地继续尝试下一个子句。例如，假设你通过某种方式在某个保护式中发送了一条消息，但这个保护式却被判为失败，你是无法撤销这条消息的——可能已经有人收到这条消息并继而做出了一些外界可见的操作，比如修改了某个文件或打印了某些文本。Erlang不允许此类情况发生，这使得保护式（以及子句）更易于推导、重排和重构。

请务必将该函数加入你的模块做做实验，先用第一个版本，然后再试试带保护式的那个。用不同的输入来检验它们的行为。我们希望你能明白，这种在不重启运行时系统的前提下便可以对Erlang函数进行实验的能力，这种交互式测试和渐进式修改的能力，可以极大地提升生产力和创造力。

2.5.4 模式、子句和变量作用域

我们再举一个模式匹配的例子，展示如何在选择子句的同时提取出感兴趣的数据。以下的函数假设你正使用标记元组来表示各种几何图形的信息：

```
area({circle, Radius}) ->
    Radius * Radius * math:pi();
area({square, Side}) ->
    Side * Side;
area({rectangle, Height, Width}) ->
    Height * Width.
```

例如，你若发起调用my_module:area({square, 5})，便会得到25。你若传入{rectangle, 3, 4}，它便返回12，以此类推。模式匹配决定该选择哪个子句，但与此同时它也将变量绑定到了数据结构中的元素上，以便各个子句的函数体能够引用这些值。请注意与早先的either_or_both函数不同，该函数各子句的顺序无关紧要，因为每次仅能匹配上一个子句；它们是*互斥*的。

绑定于函数子句首部的变量的*作用域*（或生存期）遍布整个子句，一直到该子句末尾的分号或句号。例如，在area函数中，你用不同的变量名来指代圆的半径（Radius）和正方形的边长（Side），但你乐意的话也可以管它们都叫X，因为它们分属不同的子句。另一方面，矩形的高度（Height）和宽度（Width）就必须采用不同的变量名，因为它们的作用域重合。Erlang的变量

是无须声明的——按需取用便可；但这也要求同一子句内的变量不能重名。

当变量的作用域结束时，对应的值若未被程序的其他部分用到，便会被标为垃圾，等待垃圾回收器回收。在Erlang中这也是件无须费心的事儿。

2.6 case 和 if 表达式

要是Erlang中只有函数子句这一种控制流分支方法，你就必须为程序中的每个小小的分支选择都取一个函数名。虽然理论上没什么问题，但这种做法未免也太烦人了。幸运的是，Erlang专门提供了针对这种情况的case表达式。这些表达式同样可以有多个子句，但每个子句只能有一个模式（所以无须括号）。

例如，2.5.4节的area函数也可以用case表达式来写：

```
area(Shape) ->
    case Shape of
        {circle, Radius} ->
            Radius * Radius * math:pi();
        {square, Side} ->
            Side * Side;
        {rectangle, Height, Width} ->
            Height * Width
    end.
```

请注意你必须得给area的输入参数一个名字，这样才能在后续的分支判断中引用该参数的值（case Shape of ...）。同时请注意所有的子句都以分号分隔，这点和函数子句一样，另外整个case表达式必须以关键字end结尾。（最后一个子句之后没有分号——分号是分隔符，不是结束符。）在这个案例中，由于引入了额外的变量以及case/of/end关键字，新函数的可读性可能反而有所降低，大部分Erlang程序员还是更倾向于早先版本的写法（即便要把函数名重复3遍）。

当你想在case表达式中针对多个项式进行分支选择时，你必须用元组标记将它们组合起来。例如，2.5.3节的either_or_both函数可以写成：

```
either_or_both(A, B) ->
    case {A, B} of
        {true, B} when is_boolean(B) ->
            true;
        {A, true} when is_boolean(A) ->
            true;
        {false, false} ->
            false
    end.
```

可以看到，在case表达式中也可以使用保护式（when ...）。同样，你也可以选用该函数先前的更简明的版本。

2.6.1 Erlang的布尔型if-then-else分支选择

大吃一惊吧：根本没有这种玩意儿！你可以用case表达式来替代，如下：

```
case either_or_both(X, Y) of
    true  -> io:format("yes~n");
    false -> io:format("no~n")
end
```

尽管在最后一个分支中可以拿下划线用作通配模式，不要这么做，分别在两个分支中写明true和false。当分支选择的输入碰巧是true/false之外的值时，这样做可以确保你的程序尽早报错，同时Dialyzer这类程序分析工具也可以更明了你的意图。

2.6.2　If表达式

作为特例，if表达式是case表达式的一种缩略形式，它不针对特定的值做分支判断也不含模式。当你仅依赖保护式进行子句选择时，就可以使用if表达式。例如：

```
sign(N) when is_number(N) ->
    if
       N > 0 -> positive;
       N < 0 -> negative;
       true  -> zero
    end.
```

这段逻辑也可以用带占位符的case（同时在所有子句中应用省略模式）来写：

```
sign(N) when is_number(N) ->
    case dummy of
       _ when N > 0 -> positive;
       _ when N < 0 -> negative;
       _ when true  -> zero
    end.
```

通过这个示例我们希望你能明白为何if表达式最后的通配子句（catch-all clause）要写成true -> ...。只要保护式判定为真，子句就总能匹配。

If表达式很久以前就被加到Erlang语言中了，坦白说这一设计相当随意。它们并不常用，因为大部分分支判断都或多或少地依赖于模式匹配。虽然它们在少数场合很好用，但Erlang程序员长久以来一直埋怨该表达式是对if关键字的浪费。作为初学者，你只要记住分支判断的条件并非普通表达式——它们是*保护式判定*，因而能力有限（参见2.5.3节）。

2.7　fun 函数

我们曾在2.2.8节简要介绍过fun函数。然而只有现在，在我们介绍完函数和子句之后，我们才能详细讨论如何创建fun函数。

2.7.1　作为现有函数别名的fun函数

若要引用当前模块内的某个函数——比如either_or_both/2——并告知程序的其他部分："请调用这个函数"。那么你可以创建一个这样的fun函数：

```
fun either_or_both/2
```

和各种其他类型的值一样，你可以将之与变量绑定

```
F = fun either_or_both/2
```

或直接将之传递给别的函数：

```
yesno(fun either_or_both/2)
```

对于来自别处的fun函数，你可以像调用任意普通函数一样来调用它，如：

```
yesno(F) ->
    case F(true, false) of
        true  -> io:format("yes~n");
        false -> io:format("no~n")
    end.
```

这样一来，程序的行为也可以简单地参数化。随给定的fun函数的不同，同一函数（yesno）可以发挥不同的作用。在这个示例中，输入参数F必须是一个函数，同时它要能接受两个布尔参数并返回一个用于条件判断的布尔值，除了这些限制之外，它无所不能。

高阶函数

上述示例中的 yesno/1 函数就是所谓的高阶函数：以 fun 函数为输入，或以 fun 函数为输出，或兼而有之。fun 函数和高阶函数非常有用；面向对象语言中的各种代理、适配器、命令、策略等模式都可以用它们来实现。

请注意，这些本地别名fun函数与匿名fun函数（即将阐述）在实现上很类似，它们都依赖于模块的当前版本。详情参见下一节中的“本地fun函数短暂的有效期”。

远程别名fun函数

如果要引用位于其他模块中的函数，可以采用以下语法来创建fun函数（对同一模块内的导出函数同样有效）：

```
fun other_module:some_function/2
```

远程别名fun函数在代码加载方面有着不同的行为：它们不依赖于被引用函数的特定版本。相反，在被调用时，它们总是指向被引用函数的最新版本。这些fun函数类型的值仅仅是函数的符号引用，因此可以被长期存储并/或毫无障碍地在Erlang系统间传递。

2.7.2　匿名fun函数

尽管被用作别名时fun函数很有用，但它真正的威力却在于匿名fun函数，也就是所谓的Lambda表达式。如同上一节中的fun函数，它们以fun关键字开头；同时，如同case表达式，它们也以end关键字结束。位于这两个关键字之间，是一个或多个不带函数名的函数子句。例如，以下是一个最简单的匿名函数。它不接受任何参数且总是返回零：

```
fun () -> 0 end
```

再来看个复杂点儿的函数——其功能与2.5.4节中的area函数一致，只是没有函数名：

```
fun ({circle, Radius}) ->
        Radius * Radius * math:pi();
    ({square, Side}) ->
        Side * Side;
    ({rectangle, Height, Width}) ->
        Height * Width
end
```

显然，匿名fun函数要发挥作用，就必须与变量绑定，或者作为参数被传给其他函数，正如2.7.1节中的`yesno/1`函数：

```
yesno( fun (A, B) ->  A or B  end )
```

> **本地 fun 函数短暂的有效期**
>
> 匿名 fun 函数以及用作本地函数别名的 fun 函数，都依赖于代码的特定版本。这些 fun 函数在所属模块重新加载一次以上之后便会失效：再有人企图调用它的话，它便会抛出一个异常。因此，这些 fun 函数值不适合长期保存（例如将它们存入数据库）。此外，你若将它们以消息的形式发给别的 Erlang 系统，则接收方必须持有相同版本的代码才能顺利调用这些 fun 函数。在这种场景下应该使用远程别名。

闭包

术语闭包与fun函数或Lambda表达式往往可以互换使用，但闭包一般特指`fun...end`的内部引用了，在fun函数外部绑定的变量的情况。fun函数能将那些变量当前的值封存起来。这种用法非常常见，而且很有用。

说得具体一点，假设有一个列表，其元素是成对的字符串，每对字符串表示一个名称以及对该名称的说明。另外还有一个函数`to_html`（源码不在此列出），用于将这些字符串组装成定义列表之类的HTML片段——具体做法不限。另外，针对每个名称串，但不针对描述串，`to_html`还会调用一个由你指定的回调函数，你可以在该函数中给名称串加上自定义的额外标记以示强调。该回调函数在`to_html`完成字符串的HTML转义工作之后才会执行，因此你不用关心转义之类的细节。

于是，你可以通过以下手法将名称加粗：

```
to_html(Items, fun (Text) -> "<b>" ++ Text ++ "</b>" end)
```

请注意此处的`Text`是fun函数的参数，代表的是`to_html`每次执行回调时传入的经过转义的字符串。例如，当`Items`的值为`[{"D&D", "Dungeons and Dragons"}]`时，输出如下：

```
... <b>D&amp;D</b> ... Dungeons and Dragons ...
```

（根据`to_html`的具体功能的不同，其余的HTML标记可以是任意内容：定义列表、表格、一堆`div`，诸如此类。）

现在，假设你想将用于表示强调的HTML标记也参数化：以变量的形式让程序从其他位置传入，然后在fun函数中使用该变量：

```
render(Items, Em) ->
    to_html(Items,
            fun (Text) ->
                "<" ++ Em ++ ">" ++ Text ++ "</" ++ Em ++ ">"
            end).
```

这个fun函数用到了在fun函数外部绑定的变量（这里是Em），它给这些变量当前的值做了一个快照并封存起来——这便是*闭包*一词的由来。

Erlang的单次赋值与引用透明属性可以保证这些值不被任何人修改，于是你知道，无论什么时候调用这个fun函数，它所持有的那些变量的值都与它被创建的那个时刻保持一致。（当然，你可以针对同一个fun函数创建多个*实例*，分别引用不同的外部绑定变量，但每个实例在其生存期都是完全独立的。）

上述的这个fun函数是三种不同信息源的交汇点：render的调用者，用于指定用在名称串上的标记应该是"b"还是"i"，还是别的东西；to_html函数，用于完成将条目转换为HTML的主体工作；还有render函数，用于指定如何添加额外的标记（也就是回调函数所做的事情）。注意to_html函数要求回调函数接受一个字符串参数。回调函数的接口同时也是to_html接口的一部分。

2.8　异常与 try/catch

前面我们曾经提到过异常，但没有深入解释。那么，异常到底是什么呢？你可以将之认为是函数的另一种返回形式，区别在于它不仅会返回至调用者，还会返回至调用者的调用者，并一路向上，直至被捕获或抵达进程调用的起点（这时进程便会崩溃）为止。

Erlang的异常分为三类。

- error——这类是运行时异常，在发生除零错误、匹配运算失败、找不到匹配的函数子句等情况时触发。这些异常的特点在于一旦它们促使某个进程崩溃，Erlang错误日志管理器便会将之记录在案。
- exit——这类异常用于通报“进程即将停止”。它们会在迫使进程崩溃的同时将进程退出的原因告知给其他进程，因此一般不捕获这类异常。exit也在进程正常终止时使用，这时它会令进程退出并通报“任务结束，一切正常”。无论是哪种情况，进程因exit而终止都不算是意外事件，因而也不会被汇报至错误日志管理器。
- throw——这类异常用于处理用户自定义的情况。你可以用throw来通报你的函数遭遇了某种意外（比如文件不存在或遇到了非法输入），也可以用它来完成所谓的*非局部返回*或是用于跳出深层递归。如果进程没能捕获throw异常，它便会转变为一个原因为nocatch的error异常，迫使进程终止并记录日志。

2.8.1　抛出（触发）异常

针对每种异常，都有一个与之对应的用于抛出（或触发）异常的内置函数：

```
throw(SomeTerm)
exit(Reason)
erlang:error(Reason)
```

throw和exit都是常用函数，因此会被自动导入：调用时无须加erlang:前缀。通常你不需要在代码中抛出error类异常（但在编写库时，适时抛出badarg之类的异常却是个好习惯，正如Erlang标准库中的函数一样）。

作为特例，进程调用exit(normal)所抛出的异常不会被捕获，该进程会像完成使命后寿终正寝一样终止。这意味着其他（与之链接的）进程不会将之视作反常的终止行为（其余所有的退出原因都会被视作反常）。

2.8.2　运用**try...catch**

在现代Erlang中，你可以用try表达式来处理某段代码中发生的异常。大部分情况下，它与case表达式类似，其最简形式为：

```
try
    some_unsafe_function()
catch
    oops         -> got_throw_oops;
    throw:Other  -> {got_throw, Other};
    exit:Reason  -> {got_exit, Reason};
    error:Reason -> {got_error, Reason}
end
```

位于try和catch之间的是*正文*（body）或*保护区*（protected section）。在正文内抛出并试图传播出去的任何异常都会被捕获并与catch和end之间列出的子句做匹配。如果没有匹配的子句，那么该异常会继续传播，就好像正文外围根本没有过try表达式。与此类似，如果正文在求值过程中没有触发任何异常，则正文的结果也就是整个表达式的结果，就好像try和catch...end根本不存在。唯一的区别在于，一旦异常发生且能够与某个子句相匹配，该子句的结果便会成为整个表达式的结果。

这些子句的模式有些特殊——它们可以用冒号（:）作为异常的类别（error、exit或throw）和被抛出的项式的分隔符。如果省略类别，则默认为throw。一般情况下不应该去捕获error和exit，除非你明确知道自己在做什么。这种做法违背了速错的理念，还有可能掩盖真正的症结。某些情况下，你需要运行一些不那么可信的代码并捕获从中抛出的所有东西。这时你可以使用以下模式来捕获所有异常：

```
_:_  ->  got_some_exception
```

（需要检查异常中的数据的话，可以使用Class:Term -> ...。）

另外，需要注意的是一旦进入catch部分，代码就不再受到保护。catch子句中抛出的新的异常，会传播到try表达式之外。

2.8.3　**try...of...catch**

当你需要区分正常情况和异常情况并做出不同的处理时，可以使用try的复杂形式。例如，

如果你想在正常情况下继续处理表达式的结果值，但在异常情况下打印一条错误信息并退出，那么你可以增加一个of...段，如：

```
try
    some_unsafe_function(...)
of
    0 -> io:format("nothing to do~n");
    N -> do_something_with(N)
catch
    _:_ -> io:format("some problem~n")
end
```

求值成功后你所要做的第一件事（除了给求值结果命名）一般都是以求值结果为条件做分支判断，你可以像在case表达式中那样在of和catch之间写上一个或多个子句，用于处理try...of部分求值成功后的逻辑。但要注意，of...部分和catch部分一样，是不在保护范围内的——当前的try表达式无法捕获这里发生的异常。

2.8.4　after

最后，你还可以给任意try表达式加上一个after段。其作用在于确保某段具有副作用的代码的执行。在你离开try表达式之前，无论表达式的其余部分发生了什么，after段的代码都会被执行。这种机制通常用于完成各种形式的资源释放——例如，在以下这个示例中是确保某文件的关闭：

```
{ok, FileHandle} = file:open("foo.txt", [read]),
try
    do_something_with_file(FileHandle)
after
    file:close(FileHandle)
end
```

此处，如果{ok,FileHandle}=...匹配成功，就表示文件已经成功打开。随后便进入try表达式，其中的after段会确保文件关闭，即便发生异常也没关系。

注意，有了after，catch部分就不必要了（当然你可以加上，of部分也一样）。无论哪种情况下，只有整个try表达式全部就绪之后after部分的代码才会执行，这里所谓的就绪也包括从of部分或catch的某个子句中又抛出新的异常的情况。若确实抛出了新的异常，该异常会被暂时挂起，直到after部分执行完毕后再被重新抛出。如果after部分又抛出异常，抛出的异常便会取代之前的异常，而原先被挂起的异常则会被丢弃。

2.8.5　获取栈轨迹

通常，你所见到的异常并不包含执行栈的轨迹，它被存储于内部。你可以通过调用内置函数erlang:get_stacktrace()来查看当前进程最近抛出的异常的栈轨迹。

栈轨迹（stack trace）是异常发生那一刻位于栈的顶部的那些调用的逆序列表（最后一个调用位于最前）。每个函数都被表示成{Module, Function, Args}的形式，其中Module和Function都是原子，Args要么是函数的元数，要么是函数被调用时的参数列表，这取决于当时

的可用信息的情况。一般来说，只能看到最顶层调用的参数列表。

注意，如果调用erlang:get_stacktrace()后得到一个空表，就表示直至目前为止该进程尚未捕获任何异常。

2.8.6 重抛异常

有时你可能需要进一步检查异常之后才能判断是否进行捕捉。尽管不太常用，但必要的话你确实可以先捕获异常，再通过内置函数erlang:raise(Class, Reason, Stacktrace)重新将之抛出。此处的Class必须是error、exit或throw，Stacktrace则应该来自erlang:get_stacktrace()。例如：

```
try
  do_something()
catch
    Class:Reason ->
        Trace = erlang:get_stacktrace(),
        case analyze_exc(Class, Reason) of
            true  -> handle_exc(Class, Reason, Trace);
            false -> erlang:raise(Class, Reason, Trace)
        end
end
```

在这段代码中，你可以捕获任何异常，对其进行分析，再决定是自行处理还是重新抛出。然而，这种做法既繁琐又低效（因为需要将栈轨迹转换成符号化的元组列表），只能用在万不得已的时候。

2.8.7 传统的catch

早在try表达式进入Erlang之前，catch就已经存在了。作为当时唯一的异常处理机制，catch在老代码中很常见。它是这么用的：catch Expression对Expression求值，若能够得出结果（即不抛出异常），便以此为结果。否则，若发生异常，便将之捕获并作为catch的结果，结果的格式根据异常类别的不同而不同。以下的shell交互过程演示了多种不同情况：

```
1> catch 2+2.
4
2> catch throw(foo).
foo
3> catch exit(foo).
{'EXIT',foo}
4> catch foo=bar.
{'EXIT',{{badmatch,bar},[{erl_eval,expr,3}]}}
```

简而言之，对于throw，得到的结果就是被抛出的项式；对于exit，得到的是包含退出原因的标记元组（'EXIT'是一个全部大写的原子，一般不容易误用）；而对于error，得到的是一个包含异常本身和栈轨迹的标记元组。这种设计看似简单，但实际上却把事情复杂化了，使我们难以甚至无法准确判定到底发生了什么以及如何进行后续处理。你应该避免使用传统的catch，在老代码中碰到它时能够理解得了就可以了。

2.9　列表速构

速构是一种用于描述作用于集合或元素序列（如列表）上的运算的紧凑记法。你很可能早已在常见的描述集合的数学语言中见识过它了，比如，$\{ x \mid x \in N, x > 0 \}$便是一个例子，它可以解读为“所有属于自然数集合（由N表示）且大于零的*x*值”——也就是所有正整数。

如果你还不熟悉集合的标记法，就仔细看看这个例子，很快便能明白是怎么一回事。竖线 | 用于分隔模板部分和生成器与约束条件部分，前者描述如何构建单个元素，后者指定元素来源和约束条件。在这个例子中，模板就是*x*，生成器则是“*N*中的所有*x*值”，同时还有一个约束条件，限定只有那些大于零的*x*才能进入结果集。总的来说非常简单，但的确是这类运算的一种有效的表述形式。

2.9.1　列表速构记法

Erlang是一门编程语言，而非纯粹的数学。你可以在语法中融入相同的思想，但必须更加具体。尤其是，元素的顺序[①]和选用的数据结构都显得更为重要。Erlang中用于表示元素序列的首选数据结构自然是列表，所以才有所谓列表速构。语法也要稍作修饰。例如，假设你有一个现成的整数列表，正负整数都有，通过以下方法你可以便捷地从中创建出一个仅含正整数的新列表（它们在列表中的顺序仍然保留）：

```
[ X  ||  X <- ListOfIntegers,  X > 0 ]
```

注意此处必须使用双竖线||，因为单竖线已经被用在普通列表单元上了。除此以外，我们仍然使用[...]来表示列表。由于键盘上没有∈，我们用左箭头<-来表示生成器；||右侧除生成器以外的部分便是约束条件，如X > 0。模板部分可以是任意表达式，并可以使用绑定于竖线右侧或列表速构之外的任意变量（前者如由生成器绑定的变量X）。

另外，如果在速构中指定多个生成器，便会像写嵌套循环一样产生元素的各种组合。这种用法的用处不多，但偶尔也能用得上。[②]

2.9.2　映射、过滤和模式匹配

单个列表速构可以用于完成各种映射和过滤运算的组合，其中映射是指针对元素完成一些运算后再将运算结果放入结果列表。例如，以下的列表速构能够从源列表中选出所有的正偶数（rem表示求余运算）并求出它们的平方：

```
[ math:pow(X,2) || X <- ListOfIntegers, X > 0, X rem 2 == 0 ]
```

但列表速构最强大的能力还是源自模式匹配。在生成器中，<-箭头左侧不一定是变量——可

① 此处注意数学中的集合是不强调元素的顺序的。——译者注

② 例如表达式[{X, Y} || X <- [1, 2], Y <- [a, b]]的结果是[{1,a},{1,b},{2,a},{2,b}]。——译者注

以是任意模式，跟匹配运算符（=）差不多。这意味着生成器本身就内置了一个约束条件：只有与模式相匹配的元素才在考虑范围内；其余元素统统忽略不计。此外，借助模式你还可以抽取出元素的不同组成部分并将之用在约束条件或模板中。例如，2.5.4节的area函数曾经用到一类用于表示几何图形的元组，现在假设你有一个以这类元组为元素的列表。你可以从中选出那些面积不小于10的矩形，并创建一个与之相对应的面积列表，如下：

```
[ {area, H*W} || {rectangle, H, W} <- Shapes, H*W >= 10 ]
```

你应该试着尽可能地使用列表速构。除了效率因素外，它也是这类运算最紧凑和可读的表达形式。

2.10 比特位语法与位串速构

我们曾在2.2.2节介绍过二进制串和一般位串，但我们只演示了如何创建简单的二进制串（其长度为8的整数倍——即这些二进制串可以视作完整字节的序列）。然而在现代Erlang中，位串的长度可以任意。通过比特位语法你可以随心所欲地构造指定尺寸和布局的二进制串；反之，它也可以用于匹配和抽取位串中指定的区段（例如从文件或套接字中读取二进制数据）。配合上速构，这种语法会变得极为强大。

2.10.1 构造位串

位串可以写作`<<Segment1, ..., SegmentN>>`，其中双小于号和双大于号之间可以包含零个或多个区段指示符（segment specifier）。位串以比特位为单位的整体长度，就是各区段长度的总和。

区段指示符可以为以下形式之一：

```
Data
Data:Size
Data/TypeSpecifiers
Data:Size/TypeSpecifiers
```

Data必须是整数、浮点数或另一个位串。你可以将区段长度指定为单位长度的某整数倍，你还可以指定区段的类型，该类型决定了如何解读Data以及如何对它进行编解码。例如，这样一个简单的二进制串<<1,2,3>>，它有3个区段，每个区段的数据都是一个整数，区段既没有尺寸也没有类型指示符。这个例子中，类型默认为integer，而integer的默认尺寸为1。integer类型的单位是8比特位，因此区段被编码为8位无符号字节。与此类似，<<"abc">>是<<$a,$b,$c>>的缩写——也就是一个8位整数字符编码（Latin-1）序列。整数的位数如果超出了区段的最大空间，就会被截断，所以<<254,255,256,257>>就变成了<<254,255,0,1>>。

区段的类型必须由你指定，不取决于Data自身的类型。乍一看很方便，但这样做实际上有违速错的哲学，并可能将你带入尴尬的境地——比如向文件中写入错误的数据什么的。再举个例子，你无法像这样串接两个位串：

```
B1 = <<1,2>>,
B2 = <<3,4>>,
<<B1, B2>>
```

因为默认情况下B1和B2是被当作整数的。但如果你指定B1和B2是位串，就能行得通：

```
<<B1/bits, B2/bits>>
```

这样便可以得到你期望的<<1,2,3,4>>。

你可以通过TypeSpecifiers部分（位于“/”后面）来控制区段编解码的细节。它由一个或多个由短杠（-）分隔的原子组成，如integer-unsigned-big。原子的出现次序并不重要。当前你可以使用的指示符的集合如下：

- integer、float、binary、bytes、bitstring、bits、utf8、utf16、utf32
- signed、unsigned
- big、little、native

作为特例，还可以加上unit：Integer。这些指示符可以按多种方式组合，但上述列表中每组至多只能出现一个。bits是bitstring的别名，bytes是binary的别名）。对于integer、float和bitstring类型，尺寸单位是1比特位，binary的单位则是8比特位（一字节）。

还有许多有待讨论的细节，篇幅所限，不再赘述。当你真正开始使用二进制串时，请查阅官方文档或其他Erlang编程相关的书籍。此处的内容应该足够让你理解相关问题的概念了。

位串中的 UTF 编码

Erlang 中新进加入了一个功能，令你能够将位串区段指定为 utf8、utf16 和 utf32 类型，可能你已经从前面的类型指示符列表中注意到了。这使你能够在位串中使用 UTF 编码的字符。例如：

```
<<"Motörhead"/utf8>>
```

你无法指定这类区段的长度，它们的长度是由输入决定的。在这个例子中，一共使用了 10 字节来编码这 9 个字符。

2.10.2　比特位语法中的模式匹配

正如你可以用同样的语法来构造和分解元组，你也可以用同样的比特位语法来分解位串中的数据。相较于手工完成各种位移和掩码运算，用比特位语法来解析各种怪异的文件格式和协议数据显得手到擒来，也更不容易出错。作为一个经典示例，下面将为你展示如何利用函数子句中的模式来解析IP报文首部的内容：

```
ipv4(<<Version:4, IHL:4, ToS:8, TotalLength:16,
       Identification:16, Flags:3, FragOffset:13,
       TimeToLive:8, Protocol:8, Checksum:16,
       SourceAddress:32, DestinationAddress:32,
       OptionsAndPadding:((IHL-5)*32)/bits,
       RemainingData/bytes >>) when Version =:= 4 ->
    ...
```

只要传入的报文的尺寸足够进行匹配，且Version字段为4，报文便会被解析为相应的变量，大部分变量都被解析为整数，只有OptionsAndPadding（一个长度取决于先前解析出的IHL字段的位串）和RemainingData段除外，其中后者包含报文首部之后的所有数据。从一个二进制串中抽取另一个二进制串并不涉及数据复制，因此这种运算的成本很低。

2.10.3　位串速构

存在于很多函数式编程语言之中的列表速构的思想，也被扩展到了Erlang的比特位语法中。位串速构酷似列表速构，只是[...]被换成了<<...>>。以一个小整数列表为例，所有整数都在0和7之间，你可以按每个数3比特位将它们打包成位串，如下：

```
<< <<X:3>> || X <- [1,2,3,4,5,6,7] >>
```

shell会将上式得到的位串打印成<<41,203,23:5>>。请注意末尾的23:5——位串的总长度为8+8+5=21比特位，考虑到输入列表包含7个元素，这个结果是正确的。

那么如何对这样一个位串进行解码呢？当然还是用位串速构咯！区别在于这次你需要将生成器中的<-换成<=，表示从位串中提取内容，而<-只能从列表中选取元素：

```
<< <<X:8>> || <<X:3>> <= <<41,203,23:5>> >>
```

得到的二进制串是<<1,2,3,4,5,6,7>>，由此可见，此前你确实成功地将3比特位整数格式转换成了8比特位整数格式。但若你希望结果是列表而非位串该怎么办呢？把位串生成器用到列表速构里就可以了！

```
[ X || <<X:3>> <= <<41,203,23:5>> ]
```

产生的对应的列表为[1,2,3,4,5,6,7]。我们建议你在shell中多多把玩一下比特位语法。借助比特位语法和一点点创造力你便可以完成很多有趣的事情。

2.11　记录语法

为了不让之前的章节被各种怪异语法所充斥，我们直到现在才来介绍Erlang的另一个重要组成部分：记录语法。

元组是大部分Erlang结构化数据的基石，然而从软件工程角度上看，它们还不够灵活。想象一下，在你的设计中每个客户的信息（打个比方）都由一个含有五个元素的元组来表示，你开发的整个程序（可能包含多个模块）都围绕着这个设计展开。随着业务的演化，你很可能会发现需要增加一个字段，于是你不得不跑遍所有代码并修改所有涉及这些元组的地方，既包括创建元组的地方，也包括各个针对它们做匹配的模式。更不用说在这个过程中是多么容易犯错：要是一不小心在什么地方误把五元组写成了四元组，或是在到处添加新的字段时偏偏把某个实例给忘了怎么办？为了解决这个问题（但不牺牲元组的速度和较低的内存消耗），我们引入了记录语法。

2.11.1　记录声明

记录语法让你可以使用记录，它们本质上就是标记元组，但避免了使用元组时增减字段所带来的麻烦以及必须记住各个字段在元组中的顺序的问题。使用记录时的第一要务就是写下记录声明，就像这样：

```
-record(customer, {name="<anonymous>", address, phone}).
```

该声明告诉编译器你将要使用一个四元组（3个字段加上标记），其中第一个元素总是原子customer。其他字段的顺序与记录声明中一致，因此name总是第二个字段。

2.11.2　创建记录

你可以使用以下几种语法来创建新的记录元组：

```
#customer{}

#customer{phone="55512345"}

#customer{name="Sandy Claws", address="Christmas Town", phone="55554321"}
```

记录名之前必须加上#，这样编译器才会将之与记录声明相匹配。在{...}之内，你可以选择任意字段按任意顺序进行赋值（一个都不选也行）。（编译器会按声明中的顺序为它们排序。）未赋值的那些字段将被置为默认值，即原子undefined，除非你在声明中另行指定了默认值。

2.11.3　记录的字段以及模式匹配

假定你已经将变量R绑定到了上述3个例子中的第二个。现在你可以用点分记法来访问各个字段：

```
R#customer.name        →  "<anonymous>"
R#customer.address     →  undefined
R#customer.phone       →  "55512345"
```

和之前一样，你需要明确指定记录名称来告知编译器：“将R中的元组当作一条customer记录来处理。”不过最常用的提取记录字段的手段还是模式匹配。以下函数接受一个customer记录作为输入参数并确保电话号码不为undefined：

```
print_contact(#customer{name=Name, address=Addr, phone=Phone})
         when Phone =/= undefined ->
    io:format("Contact: ~s at ~s.~n", [Name, Phone]).
```

这跟拿元组做匹配很类似，只不过不再需要关心字段的数量和顺序了。如果你在记录声明中增加字段或调整字段的顺序，只需重新编译代码，便可像往常一样正常工作。

2.11.4　更新记录字段

如前所述，在Erlang中是无法部分更新现有的数据结构的，至少不能就地更新。你只能从原数据中创建一个新的、带些许修改的副本。例如，要想更新一个四元组，你就必须创建一个新的

四元组并将无须修改的元素原封不动地复制过来。这听起来很低效，但实际上每个元素只需耗费一个字长的复制成本——浅复制。元组的创建速度很快：Erlang优化了大量小型元组和列表单元的高速创建（及回收）——它们既可以作为便签数据，也可作为永久性的数据结构。

字段更新的语法与创建新记录的语法类似，只不过你需要说明原记录的来源。仍然假定前一节的第二个customer记录被存放在变量R中。下面将创建R的一个副本并修改其中的name和address字段：

```
R#customer{name="Jack Skellington", address="Hallowe'en"}
```

一定要记住此处R本身并没有被修改，你也不能重新将R赋值为新的值；要想把更新结果绑定到变量，就必须再用一个新的名字，比如R1什么的。另一方面，如果创建R1之后程序中再也没有别处会用到R，那么R会被自动回收。（聪明的编译器有时能够发现R即将被回收，于是会直接用R来创建新的R1。）

2.11.5 记录声明应该放在哪儿

对于仅用于单个模块中的记录，一般直接将记录声明写在模块的顶部，和导出声明及其他类似声明一起放在模块首部。然而如果要在多个模块中使用同一个记录声明，做法便有所不同。你不会希望在多个源码文件中重复定义同一个记录（这样一来将难以同步修改声明），因此你应该将这些需要共享的定义放到独立的*头文件*中，供所有需要这些定义的模块读取。这一切都由预处理器负责处理，而那正是我们的下一个主题。

2.12 预处理与文件包含

Erlang的预处理器与C/C++的类似，也就是说这是个*语元*[①]级（token-level）预处理器。语元就是从源码文件中切分出来的独立词汇和符号，送交预处理器处理的便是语元序列，而非文本中的字符。这种预处理器更易于理解，但能力也会受到一定的限制。

预处理是编译过程的一部分，预处理器负责3个重要任务：*宏展开*、*文件包含*和*条件编译*。我们来依次看看这3个任务。

2.12.1 宏的定义和使用

宏由define指令定义，既可以带参数也可以不带参数，举例如下：

```
-define(PI,  3.14).
-define(pair(X,Y),  {X, Y}).
```

① 语元（token），指语法分析中不可分割的最小语法单元。在语法分析之前，Erlang编译器会借助词法分析先将源码切分成语元。该译法了取自《实用Common Lisp编程》译者田春的建议。目前译者查阅到的采用相同译法的文献仅有台湾静宜大学咨询管理系蔡奇伟的《PuTeX4.0 Big5版使用手册（Rev 1.0）》（www.cs.pu.edu.tw/~tsay/putex/doc/guide40.pdf）。——译者注

在命名上，Erlang变量和原子的命名规则都适用于宏，但习惯上常量名为大写，其余大部分宏为小写。在代码中使用宏（按其定义进行展开）时，必须加一个问号作为前缀：

```
circumference(Radius) -> Radius * 2 * ?PI.

pair_of_pairs(A, B, C, D) -> ?pair( ?pair(A, B), ?pair(C, D) ).
```

在正式编译之前，这段代码会被展开如下：

```
circumference(Radius) -> Radius * 2 * 3.14.

pair_of_pairs(A, B, C, D) -> { {A, B}, {C, D} }.
```

宏不是函数的替代品，当你所需的抽象无法用普通函数来实现时，宏给出了一条生路：比如必须确保在编译期展开某些代码的时候，或者是在语法不允许执行函数调用的时候。

1. 取消宏定义

undef指令可用于移除宏定义（前提是该宏定义存在）。例如，经由下列几行代码之后

```
-define(foo, false).
-undef(foo).
-define(foo, true).
```

foo宏最终被定义为true。

2. 常用的预定义宏

方便起见，预处理器预先定义了一些宏，其中最有用的大概就是MODULE宏了。它展开后是一个原子，对应的是当前正在编译的模块的名称。你还可以用FILE和LINE宏获悉当前正身处哪个源文件的哪一行，如下所示：

```
current_pos() -> [{module, ?MODULE}, {file, ?FILE}, {line, ?LINE}].
```

与记录声明（参见2.11节）类似，要想在多个源码文件中共享同一个宏，就必须将宏定义放入头文件。这正好将我们带入下一小节。

2.12.2　文件包含

通过使用包含指令，Erlang源码文件可以包含另一个文件，形式如下：

```
-include("filename.hrl").
```

预处理器会读取被包含文件的内容并将之插入到包含指令所处的位置。这类文件中通常只有声明，没有函数；文件包含一般都出现在模块源文件的头部，因此这些文件也被称为头文件。按惯例，Erlang头文件以.hrl为扩展名。

在查找由-include("some_file.hrl").这类指令指定的文件时，Erlang编译器会同时在当前目录中以及列于包含路径内的目录中查找名为some_file.hrl的文件。利用erlc的-I标志，或shell函数c(...)的{i,Directory}选项可以向包含路径中添加新的目录，如下：

```
1> c("src/my_module", [ {i, "../include/"} ]).
```

include_lib指令

如果你的代码依赖于别的Erlang应用或库的头文件，你就必须知道该应用的安装位置，以便

将它的头文件目录纳入包含路径。此外，安装路径中还可能含有版本号，导致应用升级后包含路径也要跟着更新。为了尽量避免这些麻烦，Erlang提供了一个特殊的包含指令：include_lib。用法举例如下：

```
-include_lib("kernel/include/file.hrl").
```

该指令会相对于Erlang系统现有应用（尤其是所有随Erlang一并发布的标准库）的安装位置来查找文件。比如kernel应用可能被安装在C:\\Program Files\\erl5.6.5\\lib\\kernel-2.12.5。于是include_lib指令会将文件名起始处的kernel/匹配至这个路径（除去版本号）并在此目录下寻找含有file.hrl的子目录include。即便Erlang升级，你的程序也不用做任何修改。

2.12.3 条件编译

条件编译就是让编译器按特定条件忽略程序的某些部分。这种手法常用于生成程序的多种版本，比如专用于调试的版本。以下预处理指令可以控制编译器在特定的时机忽略特定位置的代码：

```
-ifdef(MacroName).

-ifndef(MacroName).

-else.

-endif.
```

恰如其名，ifdef和ifndef用于测试某个宏是否被定义过。每个ifdef或ifndef，都必有一个配对的endif来标识条件编译区（conditional section）的结束。此外，还可以用else将条件编译区切为两段。例如，在以下代码中只有定义了DEBUG宏（定义成任何值皆可）才能导出函数foo/1：

```
-ifdef(DEBUG).
-export([foo/1]).
-endif.
```

你可以通过shell函数c的{d,MacroName,Value}选项或erlc命令的-Dname=value选项，在命令行下或在自己的构建系统中对此进行控制。DEBUG宏的值在此无关紧要，一般取值为true即可。

Erlang中以句号结尾的声明统称为form，Erlang的解析器便是以form为单位来工作的，条件编译不能用在函数定义内部，因为ifdef后面的句号会被解析器误判为函数定义的结束符。不过，你可以定义一个受条件编译控制的宏，然后在函数里使用它，像这样：

```
-ifdef(DEBUG).
-define(show(X), io:format("The value of X is: ~w.~n", [X])).
-else.
-define(show(X), ok).
-endif.

foo(A) ->
    ?show(A),
    ...
```

如果在编译时定义了DEBUG宏，函数foo会首先将A的值打印到终端，然后再接着执行后续的代码。否则，函数的第一个表达式将展开为原子常量ok，该常量毫无用处，编译器会通过优化将之抹除。

2.13　进程

在第1章，我们曾介绍过进程、消息，以及进程链接和信号的概念。我们还在2.2.7节描述过进程标识符（pid）。这一节，我们将讨论运用Erlang进程时应当了解的最为重要的内容。

2.13.1　操纵进程

在1.1.4节，我们演示了如何派生进程，用!运算符在进程之间发送消息，以及用receive从信箱中提取消息。那时我们尚未深入介绍模块、函数名和函数的元数，但现在你对它们已经非常熟悉了。

1. 派生和链接

进程派生函数有两个：第一个函数仅有一个参数，就是用作新进程入口的（空元）fun函数；另一个则需要模块名、函数名和参数列表3个参数：

```
Pid = spawn(fun() -> do_something() end)
Pid = spawn(Module, Function, ListOfArgs)
```

第二种方法要求给定的函数必须事先从模块中导出，且初始数据只能由参数列表传入。同时，第二种方法总会采用模块的最新版本，在调用远程机器上的函数时一般推荐这种方法，因为远程机器上的模块版本与本地版本有可能不一致。利用这种方法派生进程的示例如下：

```
Pid = spawn(Node, Module, Function, ListOfArgs)
```

此外，还有一个名为spawn_opt(...)的版本，该版本可额外接受一个选项列表：

```
Pid = spawn_opt(fun() -> do_something() end, [monitor])
```

spawn_opt(...)可识别的选项之一就是link。有一个专门针对该选项的函数用于简化调用：

```
Pid = spawn_link(...)
```

先派生进程再用link(Pid)创建链接会引入竞态条件[①]，spawn_link(...)则可以确保进程创建与进程链接创建的原子性，从而避免竞态条件。

所有这些派生函数都会返回新进程的进程标识符，通过该标识符父进程可以与新进程通信。然而新进程对父进程却一无所知，只能通过其他方式获悉相关信息。

通过内置函数self()进程可以获取自身的pid。例如，以下代码派生出的子进程知道自己的父进程是谁：

```
Parent = self(),
Pid = spawn(fun() -> myproc:init(Parent) end)
```

这段代码假设myproc:init/1是你所要启动的子进程的入口，并将父进程的ID作为唯一参数。特别需要注意的是self()必须在fun...end之外调用，否则执行调用的将是新的子进程（它并不知道自己的父进程是谁）。这就是为什么要先获取父进程的pid再通过变量将之传给子进程。（回想一

① 先派生进程再创建链接的问题在于，如果新进程在链接创建前终止，其他进程无法收到进程终止的通知。
——译者注

下我们在2.7.2节讲过的闭包，这里的派生函数拿到的就是一个即将运行在新进程中的闭包。）

2. 进程监视

链接有一个替代品，称作监视。这是一种单向链接，可以让一个进程在不影响目标进程的情况下对目标进程施行监视。

```
Ref = monitor(process, Pid)
```

由Pid标识的进程一旦退出，实施监视的进程将会收到一条含有唯一引用Ref的消息。

3. 靠抛异常来终结进程

exit类异常用于终止运行中的进程。这类异常可通过BIF exit/1抛出：

```
exit(Reason)
```

除非被进程捕获，否则该调用将令进程终止，并将Reason作为退出信号的一部分发送给所有与该进程链接的进程。

4. 直接向进程发送退出信号

进程除了在意外退出时会自动发送信号以外，还可以直接向其他进程发送退出信号。收发双方事先无须链接：

```
exit(Pid, Reason)
```

注意这里用的是exit/2，而非exit/1——它们是完全不同的函数（却不幸都名为exit）。该信号终止的不是发送方，而是接收方。如果Reason是原子kill，接收方将无法捕获该信号，从而被强制终止。

5. 设置trap_exit标志

默认情况下，一旦接收到来自相互链接的其他进程的退出信号，进程就会退出。为了避免这种行为并捕捉退出信号，进程可以设置trap_exit标志：

```
process_flag(trap_exit, true)
```

这样一来，除了无法捕获的信号（kill）以外，外来的退出信号都会被转换成无害的消息。

2.13.2 消息接收与选择性接收

接收消息的进程可以用receive表达式从信箱队列中提取消息。尽管接收到的消息严格按照抵达顺序排列，接收方仍然可以自行决定要提取哪条消息。选择性地忽略当前暂不相关的消息（比如提前到达的消息）的能力是Erlang进程通信的一个关键特性。receive的一般形式如下：

```
receive
    Pattern1 when Guard1 -> Body1;
    ...
    PatternN when GuardN -> BodyN
after Time ->
    TimeoutBody
end
```

after...段可选，如果省略，receive永不超时。否则，Time必须是表示毫秒数的整数或原子infinity。如果Time为0，receive永不阻塞。无论哪种情况，只要信箱中没有匹配的消息，

receive便会一直等到匹配的消息到达或发生超时为止，无论先后。等待期间进程将被挂起，只有新消息到来时才会被唤醒。

每次执行时，receive会先检查最老的消息（位于队列头部），像在case表达式中那样尝试将消息与子句匹配，如果找不到匹配的子句，就继续检查下一条消息。如果某个子句的模式与当前消息匹配成功，且保护式成立（如果有的话），该消息便会被移出信箱，同时与子句对应的正文将被执行。如果一直等到超时也没有找到合适的消息，超时部分的正文便会被执行，信箱则维持原样。

2.13.3　注册进程

每个Erlang系统都有一个本地进程注册表——这是一个用于注册进程的简单命名服务。一个名称一次只能用于一个进程，换言之该机制仅适用于单例进程：一般都是些系统服务，这些服务在每个运行时系统中同一时刻最多只能有一个实例。启动Erlang shell并调用内置函数registered()，你将看到如下类似的输出：

```
1> registered().
[rex,kernel_sup,global_name_server,standard_error_sup,
 inet_db,file_server_2,init,code_server,error_logger,
 user_drv,application_controller,standard_error,
 kernel_safe_sup,global_group,erl_prim_loader,user]
2>
```

真不少啊。Erlang系统本身很像是跑着一组重要系统服务的操作系统（其中一个服务甚至叫init...）[①]。用内置函数whereis可以查找当前与指定注册名对应的pid：

```
2> whereis(user).
<0.24.0>
3>
```

你甚至可以直接用注册名向进程发送消息：

```
1> init ! {stop, stop}.
```

（实验了没？这个技巧比较晦涩，它依赖于系统进程间的消息格式。这些格式没准儿哪天还会变。）

你可以用register函数注册自己启动的进程：

```
1> Pid = spawn(timer, sleep, [60000]).
<0.34.0>
2> register(fred, Pid).
true
3> whereis(fred).
<0.34.0>
4> whereis(fred).
undefined
5>
```

（注意，在这个示例中，你启动的进程将在60秒后终止，届时，之前注册的名称将自动回归到未定义状态。）

① 大部分类UNIX操作系统启动后的初始进程都叫init。——译者注

此外，要想跟位于另一个Erlang节点上的注册进程通信，可以这样做：

```
6> {some_node_name, some_registered_name} ! Message.
```


以下这种情况可以体现出注册进程的主要优点：假设某注册进程崩溃，对应的服务被重启，新服务进程的进程标识符将发生变化。此时无须将rex服务（以此为例）的新pid逐个通知给系统中的所有进程，只要更新进程注册表即可。（但在服务完成重启和注册之前，会存在一段真空期，期间该名称不指向任何进程。）

2.13.4　消息投递与信号

Erlang进程互相用!运算符发送的消息只是Erlang通用信号系统的一种特殊形式。另一大类信号就是濒死进程向与之链接的相邻进程发送的*退出信号*；还有一小部分信号对程序员不可见，诸如尝试链接两个进程时发送的*链接请求*。（试想两个分处不同机器上的进程，你就会明白为什么这些请求必须以信号的形式存在。由于链接是双向的，双方都必须知晓链接的存在。）

投递信号时，以下的基本投递保障对所有信号都成立。

- 如果进程P1向同一个目标进程P2先后发送两个信号S1和S2（不论进程在发送S1和发送S2之间做了些什么，也不论两个信号的间隔时间有多久），这两个信号将按发送顺序到达P2（如果都能到达的话）。也就是说，抛开其他不谈，退出信号绝不会掩盖进程临死前的最后一条消息，消息也绝不会掩盖链接请求。这是Erlang进程间通信的基础。
- 尽力投递所有信号。同一Erlang运行时系统内，进程之间不存在消息丢失的危险。但是，在两个依靠网络互联的Erlang系统之间，一旦网络连接断开，消息就有可能丢失（部分依赖于传输协议）。连接恢复后，有可能出现上例中的S2最终抵达但S1却丢失的情况。

大部分情况下，你不用太关注消息的时序和投递成功率——系统的行为与你的预期基本一致。

2.13.5　进程字典

作为自身状态的一部分，每个进程都有一个私有的*进程字典*，这是一个可以用任何值作为键的简单哈希表，用于存储Erlang项式。通过内置函数`put(Key, Value)`和`get(Key, Value)`可以从中存取项式。我们不打算详细介绍进程字典，在此我们只想告诉你，无论进程字典看起来多么诱人都不要去碰它。

- 最简单的原因在于它让程序的行为难以理解。程序的行为无法再直接通过代码看出，相反，你不得不先找出执行代码的进程并搞明白该进程当前的状态才能得出准确的结论。
- 更重要的是它令进程间迁移任务的难度增加甚至趋于不可能。引入进程字典前你可以在一个进程完成一部分工作后，将剩下的任务交由另一个进程完成。现在除非第一个进程将字典数据打包发送给第二个进程，否则新进程拿不到正确的字典数据。
- 在开发库的时候，如果将某些来自不同客户端调用的信息存入进程字典（很像是Web服务器使用cookie），客户端就被迫只能使用单个进程来处理会话。如果客户端用别的进程来调用你的API，将缺少必要的上下文。

在某些情况下进程字典也有合理的用法，但一般而言，总会有更好的数据存储方案（需要的话甚至可以在不同进程间共享信息）。这种方案的名字比较怪，叫作ETS表。

2.14　ETS 表

ETS代表Erlang项式存储（Erlang Term Storage）。所谓ETS表就是一张用于存储Erlang项式（即任意Erlang数据）且可以在进程间共享的表。不过这不是违背了基本的引用透明性和避免共享的原则了吗？难不成我们走了后门偷偷摸摸地干起破坏性更新的勾当来了？四个字：进程语义。

2.14.1　为何ETS表被设计成这样

Erlang ETS表背后的基本设计思想就是，尽量让它们的接口和行为与独立的进程一般无二。即便用进程重新实现ETS表，也无须对接口做任何修改。但实际上，ETS是作为Erlang运行时系统的一部分用C实现的，既轻量又快速，接口函数也都是BIF。ETS表的性能之所以受到如此的关注，原因在于ETS是Erlang中很多东西的基础。

你仍然应该尽可能地避免数据共享，尤其是要避免那种出其不意地在别人背后修改数据的行为。然而，如果一种存储机制是通过常规进程语义和消息传递来实现的，就不会有什么根本问题，可以放心大胆地使用。更不用说这原本就是你迟早要用到的东西：用于数据存储的高效哈希表。

ETS表很像是简化了的数据库服务器：与外界隔离，并持有一些供多方使用的数据。相较于Java、C或其他类似语言中的数组，其区别在于使用ETS的客户端明确知晓它们是在跟一个有自己的生存期的实体打交道，并且从同一张表中读到的内容也可能随时间的不同而不同。但与此同时，它们也可以确信自己之前读取到的内容绝不会被偷偷摸摸地修改。在表中查找一个条目，你就会得到该条目下当前存储的元组。即便有人旋即将表中同一位置更新成了新的元组，你读到的数据也不会受到影响。相比较而言，如果是在Java数组中查找对象，则随后出现的其他线程很有可能会在查找同一个对象的同时，以某种会对你造成影响的方式修改这个对象。在Erlang中，我们力图明确区分何时引用的是会随时间变化的数据，何时引用的是简单的不可变数据。

2.14.2　ETS表的基本用法

标准库中的ets模块可用于创建和操控ETS表。要创建新表，可以调用`ets:new(Name, Options)`函数。其中名称`Name`必须是原子，`Options`必须是列表。除非设置`named_table`选项，否则名称不起实际作用（因此可以创建多张同名表）；然而对于系统调试来说，在碰到诡异的不知潜伏在何处的表时，表名可以有效地辅助定位，因此在命名时最好采用更有意义的名称，比如当前模块名，而不要用table或foo之类毫无益处的名称。

`ets:new/2`会返回一个表标识符，用于完成针对新创建的表的各种操作。例如，以下代码将创建一张表并向表中写入两个元组：

```
T = ets:new(mytable,[]),
ets:insert(T, {17, hello}),
ets:insert(T, {42, goodbye})
```

ETS表和数据库的另一个相似点在于，它同样只存储数据行——也就是元组。存储任何Erlang数据之前，都要先将之放入元组。其原因在于ETS会将元组中的一个字段用作表索引，默认采用第一个字段。（可以通过建表参数调整。）这样你便可以按第一列元素在表中查找数据行：


```
ets:lookup(T, 17)
```

这将返回[{17, hello}]。慢着，为什么结果被放在列表中呢？嗯，ETS表不一定非要是数据行的集合（所有的键都唯一），这只是默认配置；表也可以是bag（允许多个行具有相同的键但不允许出现完全相同的行）甚至duplicate bag（允许出现完全相同的行）。在这些情况下，查找结果可能不止一行。但无论是哪种情况，查不到匹配的行时都会返回空表。

有关ETS表还有很多东西要学。建表时你可以指定很多参数，还有大量用于搜索、遍历的强劲接口，以及很多。后续我们还会在第6章接触它们。

2.15　以递归代替循环

可能你已经注意到了，除列表速构以外，这门语言别无其他迭代结构。原因在于Erlang靠递归函数调用代替了迭代。虽然与我们惯用的手法不太一样，但其中并无奥妙。不过你仍然有必要了解一下相关细节和技术，这些知识可以在你还不习惯这种思维方式时帮助你更轻松地学习，同时也有助于避免常见错误从而写出坚实的代码。

入门期间，先来点儿简单的，不妨来算算0到*N*的累加和。这个问题描述起来很简单：要计算0到*N*的累加和，只要先算出0到*N*–1的累加和，再加上*N*即可。如果*N*本身就是0，累加和就是0。写成Erlang函数就是（请将之加入my_module.erl）：

```
sum(0) -> 0;
sum(N) -> sum(N-1) + N.
```

简单得不能再简单了，是吧？从来没用过递归？没关系，这个概念非常自然。（反倒是那些夹杂着break和continue的嵌套for循环之类的老古董——那些往往才是让人绞尽脑汁的东西。）但为了掌握各种迭代算法的递归写法，我们仍然有些基础知识要学。这些内容很关键，还请少安毋躁。

2.15.1　从迭代到递归

所有递归问题都可转换为相应的迭代形式（但需要你自行处理一些簿记工作）。究竟选择哪种方式，这取决于你所使用的编程语言的支持力度以及最终结果的效率。有些语言，比如各种老式Basic方言、Fortran-77或机器汇编语言，完全不支持递归。Pascal、C/C++、Java等很多语言支持递归，但由于实现层面的限制和低效，递归的作用难以得到发挥。Erlang则不同：它仅靠递归就可以创建循环，而且没有效率问题。

1. 老式循环

有时，你会选择从迭代型代码着手（也许只是在脑海中构思），再尝试换用Erlang来实现。下面这段用C或Java写成的计算0到n的累加和的代码就很典型，其中n为输入参数：

```
int sum(int n) {

  int total = 0;
  while (n != 0) {
      total = total + n;
      n = n - 1;
  }
  return total;

}
```

这段代码展示了一种过程式的算法表达方式：采用这种方法，你可以在脑海里逐步跟踪程序的执行并观察状态的变化，直至程序结束。

但是请考虑一下，要想用大白话尽可能准确地向别人描述这个算法又该怎么说呢？可能你会这么说：

(1) 整数N已知，令Total为零；

(2) 当N不等于零时，

　a) Total加N

　b) N减1

　c) 重复第(2)步

(3) 结束，Total即为最终结果。

2. 函数式循环

现在，来看看第(2)步的另一种表述方式。

若N不等于零，代入下列新值重复本步：

　a) 将Total更新为Total+N；

　b) 将N更新为N-1。

这样一来，第(2)步就变成了一个以变量N和Total为参数的递归函数。该函数不依赖其他任何信息。每次递归调用时，只需要传入新的参数值促成下次迭代，原参数值则可以直接抛弃。人们很容易通过“用不同的参数值重复同一步骤”的说法来理解迭代。（小孩子理解起来通常没什么问题；反倒是我们这些多年浸淫于过程式编程的大人会绞尽脑汁。）我们来看看这一步写成Erlang是什么样子：

```
step_two(N, Total) when N =/= 0  ->  step_two(N-1, Total+N).
```

一看就懂，是吧？（请将这个函数加到my_module.erl中。）请注意，可千万别写出N=N-1这类用法——这在Erlang中行不通：一个数无法等于它自身减1。你只能说“以N-1为新的N、以Total+N为新的Total来调用step_two”。不过好像还漏了点儿什么？任务完成后该怎么办呢？我们给这个函数再加一个子句（同时也换一个更贴切的名字）：

```
do_sum(N, Total) when N =/= 0 -> do_sum(N-1, Total+N);
do_sum(0, Total) -> Total.
```

实际上就是将上面第(3)步的文字描述翻译成了代码。一旦第一个子句匹配不上（保护式返回false），第二个子句便会生效，该子句只需直接返回Total中的结果即可。

3. 循环的初始化

现在该处理第1步了：将Total置为初值0。很明显只要在某处调用do_sum(N, 0)即可。但是在哪儿呢？实际上还有一个第零步，也就是对问题本身的描述："计算0到*N*的累加和。"也就是do_sum(N)，对吧？要计算do_sum(N)，只需计算do_sum(N, 0)，也就是

```
do_sum(N) -> do_sum(N, 0).
```

请注意一下你刚完成的事情：你创建了一个只有一个参数的函数，名为do_sum/1（注意定义末尾的句号）。该函数调用了具有两个参数的do_sum/2。回想一下，在Erlang中这是两个完全不同的函数。该例中，参数较少的那个被当作前端，另一个则不应该直接暴露给用户。因此，只有do_sum/1才应该出现在模块的导出列表中。（我们将这两个函数命名为do_sum是为了避免跟本节开头的sum函数重名，你应该把sum/1和do_sum/1都加进模块，再看看针对同一个N它们能否得出相同的结果。）

4. 终曲

我们来总结一下这两部分实现，同时做出一些改进：

```
do_sum(N) -> do_sum(N, 0).

do_sum(0, Total) -> Total;
do_sum(N, Total) -> do_sum(N-1, Total+N).
```

看明白怎么回事了吗？在上一版的递归函数中我们不过是逐字逐句地将"*N*不等于零"翻译成了代码，而在这个版本中我们调整了子句的顺序，这样一来每次调用都会先对*基准情况*（不涉及递归调用的子句）进行判定。就这个算法而言，这种手法进一步简化了代码：第一个子句试图将N与0匹配。若匹配失败，N就不会是零，此时再转入第二种情况就无须保护式了。

这一节可真够长的，但还没完，为了帮助你理解一些要点，我们还有一些功课要做，并且还要给你刚才运用过的技巧命名。首先我们来讨论一下sum和do_sum所用的两种递归。

2.15.2　理解尾递归

递归调用可分为两类：*尾递归*和*非尾递归*（有时也叫做body recursion）。本节开头的sum函数就是一个非尾递归函数（因为含有非尾递归调用）。在许多其他语言中，你能接触到的只有这种递归，因为其他情况下的递归往往可以用各种循环结构来代替。

从循环迭代中推导来的do_sum/2则是*尾递归函数*。其中所有递归调用都是*尾调用*。尾调用很容易识别，编译器总能从代码中准确地定位尾调用，并做出一些相应的特殊处理。

二者之间区别何在呢？如果是尾调用，函数在调用完成后就无事可做了（除了返回）。比较一下sum和do_sum这两个函数的函数体：

```
sum(N) -> sum(N-1) + N.

do_sum(N, Total) -> do_sum(N-1, Total+N).
```

在sum中，sum(N-1)调用完成后，在返回之前还有一些未尽事宜：即加N。另一边，在do_sum中，do_sum(N-1, Total+N)调用完成后便万事大吉——这次递归调用的返回值就是整个函数的返回值。*凡是符合这个特征的调用就是尾调用*，或称“*最终调用*”。尾调用与递归与否（调用方和被调用方为同一个函数）无关——尾递归只是尾调用的一个特例，但同时也是最为重要的一种。你能找出位于sum函数体中的尾调用吗？（没错，就是+。）

尾调用优化，值得信赖

你大概知道，每个进程在幕后都有一个栈，用于跟踪程序执行过程中的那些后续还需要回过头来处理的东西（比如说“记着一会儿还要回到这个地方来加*N*”）。栈是一种后入先出的数据结构，就像一摞一份压着一份的执行记录，当然，如果你一直记录新的内容，最终会耗尽内存。这将非常不利于程序的持久运行，那么Erlang是怎么仅靠递归来实现循环的呢？每次调用不是都要往栈上写入新的内容吗？答案是否定的，因为Erlang采用了*尾调用优化*。

尾调用优化的意思是，当编译器识别出尾调用（函数返回前的最后一个任务）时，会生成一段特殊的代码，这段代码会在执行尾调用*之前*从栈中扔掉当前调用的所有信息。此时当前调用基本无事可做，只需告知被调用的函数后续即将发生一次尾调用：“嘿！完事儿的时候直接把结果告诉我的调用者就行了，我收工了哦。”因此，尾调用不会导致栈的膨胀。（作为特例，调用相同函数的尾递归调用可以重用栈顶的调用信息，省得扔掉之后还要再重新创建。）本质上，尾调用只是“在必要时清理一些东西，然后执行跳转”。

正是基于这个原因，尾递归函数即便不停不歇地运行也不会将栈空间耗尽，同时还能达到和while循环一样高的效率。

2.15.3　累加器参数

比较前面的sum和do_sum的行为，可以发现对于同一个数N，sum将工作拆成了两半，一半负责从N倒数到零，同时在栈上记录后续要累加的数值；另一半则负责从栈记录中找出相应的数值并执行累加直至栈被清空。另一边，do_sum只在栈上保留一份记录，但它会不断更新该记录直至N递减为零。随后这份记录可以直接丢弃，同时Total成为最终的返回值。

在这个例子中，Total扮演了*累加器参数*（accumulator parameter）的角色，用于在单个变量中完成信息累加（而不是将信息记在栈上回头再来取）。编写尾递归函数时，往往需要至少一个这样的额外参数，有时还会涉及多个。这些变量必须在循环启动时初始化，因此你需要两个函数，一个用作前端接口，一个用作主循环。最终，用于保存循环过程中的临时信息的参数将被丢弃，其他参数则会成为最终返回值的一部分。

2.15.4　谈谈效率

一般来说尾递归方案比对应的非尾递归方案更为高效，但不绝对，这取决于具体的算法。非尾递归函数比较懒，只会将栈和各种需要后续回过头来处理的东西扔给系统，尾递归版本则老老实实地将达成任务目标所需的各种信息记在累加器变量中，这些信息一般被组织为列表等数据结

构。如果把非尾递归函数比作靠扔纸片作为路标才回得了家的醉汉，尾递归函数就是将所有家当都装进推车里的旅者。如果所需的最终结果只是一个数值，就像sum/do_sum的例子那样，那么旅者会大大胜出，因为负担不重跑起来很轻松。但如果结果需要追踪的信息很复杂，跟醉汉无偿获得的信息量差不多，旅者就必须自行完成一些复杂的数据管理工作，导致最终反而可能慢上一拍。

总之，有些问题用非尾递归函数解决起来更直接，有些则明显应该采用尾递归。作为脑力练习，不妨两种方法都试试，但对成品代码而言，我们建议你在尾递归和非尾递归实现中选择更可读、更易维护、你更有把握正确实现的方式。事成之后，就去干点儿别的吧。别把时间浪费在过早的优化上，尤其是以可读性为代价的优化。

当然，很多情况下其实没有什么选择余地：用到无穷循环的函数就只能采用尾递归。我们称这类函数在运行时空间消耗恒定（constant space），也就是说，即便函数永不返回，其内存占用量也不会随时间的流逝而增加。

2.15.5　编写递归函数的窍门

尝试用递归编程解决问题时，新手往往会觉得头脑一片空白——感觉根本不知道该从何处下手。下面就教你一些入门的方法。

作为示范，我们找了一个你总要以各种方式去解决的具体问题：数据结构遍历。此处我们考察的是列表，但其思想同样也适用于各种递归数据结构，比如由元组构成的树。你的任务是*反转*一个列表，或者说，针对给定列表（长度任意）创建一个逆序版本的新列表。要解决这个问题，势必要遍历原有列表中的所有元素，因为它们都会在结果中出现。

1. 参看样本

如果不知道该从何处下手，你可以先罗列几个简单的输入样本及其对应的结果。针对列表反转，可以罗列出下列样本：

```
[]        → []
[x]       → [x]
[x,y]     → [y,x]
[x,y,z]   → [z,y,x]
```

这可不是手到擒来嘛，没错，不过要想看出递归模式，把它们写下来要比全靠脑子想更为容易，也可以让问题更加具体。它还能促使你多考虑一些特殊用例。如果喜欢测试驱动开发，你还可以立即将这些用例写入单元测试。

2. 基准情况

接下来，你应该写下基准情况并明确在这些情况下都应该发生些什么。（基准情况就是那些不涉及递归调用的情况。通常基准情况只有一个，但有时也会出现多个。）对于列表反转问题，你可以将上述样本的前两个用作基准情况。不妨给你的新函数rev/1（请放入my_module）写两个子句看看：

```
rev([])  -> [];
rev([X]) -> [X].
```

这两行代码离竣工还差得远，但借助它们你至少可以立即着手测试一些简单用例，比如rev([])、rev([17])、rev(("hello"))和rev([foo])。列表元素的类型不重要，我们只在乎顺序。

走完这一步，好戏才开演：这回你必须搞定递归。

3. 数据的形态

现在开始分析其余的情况以及它们的构造方法。如果一个列表不符合上述两种基准情况，那么可以肯定它至少含有两个元素——也就是说，它会呈现为[A, B, ...]的形态。我们知道，列表是由形如[...|...]的列表单元构成的（参见前面的2.2.5节）。单独罗列出每个单元，列表就会呈现出如下的形态：

```
[ A |  [B | ...] ]
```

换言之，每个元素都占用一个独立的单元。不妨将它用作你的rev函数的第一个子句（试一下）：

```
rev([A |  [B | TheRest]  ]) -> not_yet_implemented;
```

回忆一下，函数的首部是一个模式，用于实参的匹配和分解（参见2.5.4节）。你会看到一些针对A、B、TheRest等无用变量的警告，子句的正文什么也没干，仅仅返回了一个用于说明“功能尚未实现”的原子。但至少rev函数现在已经可以接受包含两个或两个以上元素的列表了。

接下来，你得想清楚怎么处理这种情况。很明显，这里会涉及针对rev的递归调用。

4. 假设有现成的函数可用

如果从数据的样本和结构中仍然找不出头绪（别担心，勤加练习后情况就会好很多），或者明明已经看到了轮廓但就是敲不定细节，那么教你一个窍门，告诉自己：“我已经有一个现成的可工作的函数了，就在那儿，我只是要写一个新的（更好的）完成同样功能的函数。”在编写自己的新函数时，准许你采用老的版本来做试验。

不妨假设：老版本的函数名为old_rev/1。好！要把not_yet_implemented替换成更有意义的东西，你该怎么办？有变量A、B和TheRest在手，你想要一个包含同样元素的列表，只是顺序相反。如果能（用old_rev）反转TheRest，再在列表末尾加上B和A（++可用于拼接两个列表），岂不就成了？就像这样：

```
rev([A |  [B | TheRest] ]) -> old_rev(TheRest) ++ [B, A];
```

够简单的吧！现在，你的函数看似已经可以正确处理任意长度的列表了。既然它已经可以完全胜任，定然不比old_rev差，那么就换上你自己的rev吧！于是整个函数变成了这样：

```
rev([A |  [B | TheRest] ]) -> rev(TheRest) ++ [B, A];
rev([])  -> [];
rev([X]) -> [X].
```

试试my_module:rev([1,2,3,4])，一切正常，漂亮！下一步，我们来看看如何证明这个函数的确适用于所有列表。

5. 可终止性证明

有些函数我们一眼就能看出它迟早会结束——无论什么输入都不会让它陷入死循环。但对于更复杂的函数，我们很难看出它最终是否一定能在某处返回。

论证函数可终止性的主要线索就是*单调递减参数*。它的意思是，假设基准情况对应于函数可接受的最小输入，而递归情况处理其他所有较大的输入，那么，只要每种递归情况都在缩小递归调用的输入，你就知道输入参数最终一定能归结到基准情况，于是函数*必然*可以终止。但若某个递归调用传入了相等甚至更大的输入，就很值得怀疑了，必须仔细考察。（对于在多个参数上递归的函数，至少要有一个参数在其他参数不递增的前提下递减才行。）当然，像累加器参数这种与循环判定条件无关的参数可以忽略。

在`rev`的例子中，参数不可能比基准情况更小。再看递归情况，可以看到在递归调用`rev(TheRest)`时，`TheRest`所包含的元素数少于当前输入参数中以A、B开头的列表的元素数。由此可得，要处理的列表在相继缩短，于是你便知道自己绝不会陷入死循环。

针对数值做递归时，正如本节开头处的`sum`的例子，人们很容易忘记整数没有下限这个事实。回过头来观察`sum`的定义，可以看到在递归情况下传给下次递归调用的数值总是小于当前输入，因此输入参数最终必然会递减为零或更小的值。但如果传入的参数一开始就小于零，比如调用`sum(-1)`，函数就会开始用−2、−3、−4等持续不断地调用自己，直至内存被某个巨大的负数耗尽为止[①]。为了避免发生这种情况，你可以给递归情况加上保护式`when N > 0`，从而确保在输入参数为负数时无法命中任何子句。

输入参数是大还是小，是由函数的上下文决定的。例如让整数N从1递归到100，这时最“小”情况就是100，同时`N+1`“小于”`N`。关键在于每次递归调用都能向基准情况迈进一步。[②]

6. 基准情况最小化

尽管多个基准情况的存在并不会妨碍功能实现，但这会困扰后续的代码维护人员。如果真的碰到不止一个基准情况，不妨试试看能否干净利落地进行一些裁剪。在上面的例子中你罗列了两个基准情况：`[]`和`[X]`，因为这两种情况看起来最简单。但如果将`[X]`也看作列表单元，便会发现它可以写作：

```
[X | []]
```

由于`rev`已经可以处理空表，于是`rev([X])`可以归结为`rev([]) ++ [X]`。看起来好像没什么必要，但这样一来你就不用将包含单个元素的列表和包含两个或两个以上元素的列表区分为两种独立的情况来考虑了。两条规则合二为一，便得出了一个更为清晰的方案：

```
rev([X | TheRest]) -> rev(TheRest) ++ [X];
rev([]) -> [].
```

（注意此处子句的顺序并不重要：列表要么空要么非空，只有一个能够命中。但我们没必要先检查空表，对于含有100个元素的列表，前100次递归调用处理的都是非空表，空表只会出现一次。）

① 此处内存确实会被撑爆，但不是源自某个巨大的负数，而是因为sum的栈空间被撑爆。——译者注

② 这几个段落中的“大”、“小”概念用得比较含糊。作者的意思应该是将基准情况所覆盖的输入当作“最小”输入，同时越接近基准情况的输入越“小”。这里的“大”和“小”大致可以认为是在形容输入参数相较于基准情况的规模或复杂度。——译者注

7. 识别平方复杂度的行为

现在，你已经得到了一个正确的列表反转函数。万事大吉了吗？还没呢。如果输入的列表很长，这个实现的耗时也会相当长。为什么呢？因为你已然踏入了平方复杂度[①]行为的雷池。假设函数处理某个列表的耗时为T个单位（采用什么时间单位都可以），平方复杂度意味着处理长度为原列表两倍的列表将耗时$4T$个单位，处理3倍长度的列表将耗时$9T$个单位，以此类推。列表比较短小时还没什么感觉，但情况很快就会失控。假设你有一个函数，它会把给定目录下的所有文件都纳入一个列表然后进行处理。函数功能正常，但你处理过的目录所包含的文件数从来就没有超过100个。处理100个文件只需1/10秒，问题不大。然而如果采用了平方时间复杂度的算法，某天你把它用在某个包含10 000个文件的目录上时（足足是以前的100倍），耗时将会是以前的$100 \times 100 = 10\ 000$倍（超过15分钟）。与此同时，采用耗时与输入规模成正比的算法，或者说是线性复杂度的算法，只要10秒钟。客户很生气，后果很严重。

为什么你实现的`rev`会有平方级的时间复杂度呢？因为在每次递归调用中（每个列表元素一次），你都用到了`++`运算符，而它的执行时间与运算符左侧的列表长度成正比（参见2.2.5节）。假设左侧列表长度为1时`++`的耗时为T。在你的`rev`函数中，位于`++`左侧的是递归调用返回的列表，其长度与输入列表相同。也就是说在长度为100的列表上执行`rev`，第一次调用会耗时$100T$、第二次$99T$、第三次$98T$，以此类推，直到递减为1。（每次调用还会耗费一小部分时间用于切分`x`和`TheRest`并执行递归调用，但与$100T$相比，这部分时间可以忽略不计。）

$100T+99T+98T+\cdots+2T+1T$是多少呢？这相当于是在求边长为100的等腰直角三角形的面积：其面积应为$100 \times 100/2$，也就是边长为100的正方形的面积的一半。总之，处理长度为N的列表，`rev`的耗时与$N \times N/2$成正比。耗时如何随N的不断增大而增长才是我们最为关心的内容，因此我们称这个算法呈*平方复杂度*，因为耗时呈现出了$N \times N$规模的增长。与函数的主体行为相比，身为分母的2实在是回天乏术（见图2-3）。

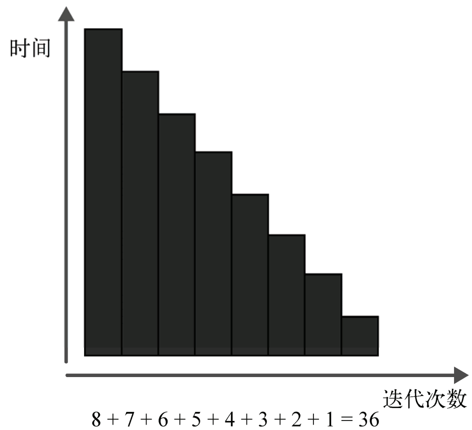


图2-3　平方复杂度的函数迭代N次后的总耗时 = 三角形的面积

① 此处及本节后续内容中提到的复杂度指的都是时间复杂度。——译者注

需要注意的是，这类由于在某数据集上进行循环或迭代而引发的问题在所有语言中都有可能发生；这不是递归的错，而是因为你将某个任务重复了*N*次，还每次都额外干些耗时与*N*成正比的事情，这样一来所有耗时加起来就变成了这么一个三角形。唯一的安慰是这还不是最坏的情况：如果你在上个例子中采用一个时间复杂度达到立方级别的算法，那就乖乖坐等28个小时吧。要是再演变成指数复杂度，可就不仅仅是等的问题了。

8. 避免平方级别的时间复杂度

怎样才能让rev函数避开平方时间复杂度呢？反正肯定不能继续把变长列表放在++运算符的左侧了。尾递归能不能解决问题呢？同样是列表遍历，但现在每走一步都将手头所需的家当悉数摆进面前的参数里，这样一来遍历一结束就万事大吉了：栈上干干净净，没有需要回过头去处理的东西，就像2.15.1节的do_sum一样。按这种方式来对列表做递归如何？你仍然可以采用编写rev时分析出来的基准情况和递归情况，不过新函数将改名为tailrev，另外还要增加一个累加器参数用于保存最终结果，如下：

```
tailrev([X | TheRest], Acc) -> not_yet_implemented;
tailrev([], Acc) -> Acc.
```

现在来看not_yet_implemented部分：要构成尾递归，一定是tailrev(TheRest, ...)的形式，且第二个参数跟Acc和X有关。你已经掌握了价廉物美的cons运算（将元素加至列表左侧），你也明白Acc将会成为最终的逆序列表。那么不妨用[X | Acc]将元素加到Acc的左侧，实际上就是在遍历的同时把老列表从右向左再写一遍：

```
tailrev([X | TheRest], Acc) -> tailrev(TheRest, [X | Acc]);
tailrev([], Acc) -> Acc.
```

这段代码在遍历列表的同时将眼前的每个元素都加到Acc的左侧，直至列表为空。那么Acc的初值应该是什么呢？要搞明白这一点，最简单的方法就是看看在基准情况以外的最简单的情况下函数会有什么行为。不妨看下这个调用：tailrev([foo], Acc)，反转一个仅含一个元素的列表。该调用可以命中第一个子句，于是X和TheRest被分别绑定为foo和[]，子句的正文也就变成了tailrev([], [foo | Acc])。下一步的递归调用就归结到了基准情况tailrev([], Acc)，并最终返回Acc。换言之，Acc的初值必须是个空表，这样[foo | Acc] = [foo]才能成立。最终的完整实现如下，其中tailrev/1是主入口：

```
tailrev(List) -> tailrev(List, []).

tailrev([X | TheRest], Acc) -> tailrev(TheRest, [X | Acc]);
tailrev([], Acc) -> Acc.
```

为什么这个实现就是线性（与列表长度成正比）而非平方时间复杂度呢？因为处理列表中每个元素所需的耗时都是固定的（诸如将元素加到列表左侧）；因此，若列表含有*L*个元素，总耗时就是*L*乘以*C*，其中*C*是某个较小的常数，而且在输入规模增大时这个算法的耗时也绝不会像平方复杂度的版本那样，让你眼睁睁地看它极速爆炸。

9. 小心长度

在Java这类语言中，获取列表的长度是一个常数时间复杂度的操作。有这类语言背景的Erlang新手经常会犯的一个错误就是在保护式中调用内置函数length，如代码清单2-2所示。

代码清单2-2　别这样做：length(List)会遍历整个列表！

```
loop(List) when length(List) > 0 ->
    do_something;
loop(EmptyList) ->
    done.
```

写成这样的函数，耗时都与列表长度的平方成正比，因为每回计算列表元素数目时都要从头至尾遍历整个列表。所有的耗时加起来又形成了一个三角形，跟上一小节中++运算符的耗时情况类似。但如果只关心列表是否为空，采用模式匹配即可，如代码清单2-3所示。

代码清单2-3　这样做才对：用模式匹配对列表判空

```
loop([SomeElement | RestOfList]) ->
    do_something;
loop([]) ->
    done.
```

这种匹配操作很快，而且耗时是固定的。你甚至可以利用模式匹配来辨识不短于指定长度的列表，如代码清单2-4所示。

代码清单2-4　用模式匹配检查不同长度的列表

```
loop([A, B, C | TheRest]) -> three_or_more;
loop([A, B | TheRest]) -> two_or_more;
loop([A | TheRest]) -> one_or_more;
loop([]) -> none.
```

注意先检查较长的列表，要是先检查长度大于等于2的列表的话，长度为3的列表也会命中。总之记得把最特殊的模式放到最前面就是了。

2.16　Erlang 编程资源

为了学习更多的Erlang语言的知识，更好地抓住函数式编程和并发编程的精髓，并学习更多的库和工具，我们为你准备了以下一系列重要资源。

2.16.1　图书

除本书以外，近年来还有两本Erlang编程语言方面的书籍。其中第一本曾在全世界范围内掀起了一波新的Erlang浪潮，它就是Joe Armstrong的*Programming Erlang——Software for a Concurrent World*[①]（Pragmatic Bookshelf，2007）。这本书对Erlang语言和并发编程做了一个很好的概要性介绍，书中还给出了大量新奇有趣又易于用Erlang实现的程序示例。

第二本的年份更近一些，它就是Cesarini和Thompson的*Erlang Programming*（O’Reilly，2009）。这本书更加深入地介绍了Erlang语言的各种细节和惯例、函数式编程和并发编程的各种技术，以及Erlang生态系统中不可或缺的各种库和工具。

① 本书中文版《Erlang程序设计》已由人民邮电出版社出版，前面也有提及。——编者注

最后，如果对Erlang的历史感兴趣，你还可以看看Armstrong、Virding、Wikstrom和Williams合著的*Concurrent Programming in Erlang*, 2nd ed.（Prentice Hall, 1996）[①]，不过这本书的内容在语言特性、库和工具等方面都已经过时了。

2.16.2 在线资料

Erlang的主站点是www.erlang.org，在这儿你可以下载到Erlang的最新开源版本、阅读在线文档、查阅来自爱立信OTP开发团队的官方新闻、订阅邮件列表等，不一而足。

还有一个社区站点www.trapexit.org，提供了邮件列表合集和wiki服务，其中囊括了教程、技术文章、导引、参考链接等一系列内容。另外，www.planeterlang.org上还汇总了四面八方的各种Erlang相关的feed，助你随时洞悉最新态势。

Erlang的主邮件列表是erlang-questions@erlang.org，在这儿，即便是最为艰深的问题往往也能得到资深专业用户的解答。合集中十几年的沉淀更是无价的信息宝库。

最后，在www.stackoverflow.com上搜索一下"erlang"，在寻求各种问题的解答时，这些内容可以作为erlang-questions邮件列表的绝佳补充。

2.17 小结

本章涵盖了大量内容，从Erlang shell开始，到数据类型、模块和函数、模式匹配、保护式、fun函数和异常，再到列表速构、比特位语法、预处理器、进程、ETS表、递归等。尽管在Erlang程序开发方面还有很多东西要学，但本章所涵盖的内容已经足以助你迈出坚实的一步。如果你还是Erlang新手，略过这章也没关系；需要时随时回头查阅即可。

在后续的章节中，我们将假设你已经掌握了前面的知识，并以此为基础循序渐进地阐述各种新的概念。现在我们将直接深入OTP，它将是本书后续内容的主线，而作为Erlang程序员的你也会将它视作忠实的伙伴（我们衷心希望如此）。

---

① Prentice Hall公开了*Concurrent Programming in Erlang*, 2nd ed. 的第一部分。这部分书稿由译者于2009年组织CPiE-CN志愿译者团队翻译完成，参见http://cpie-cn.googlecode.com/hg/_build/html/index.html。——译者注

第 3 章

开发基于TCP的RPC服务

本章概要

- 介绍OTP行为模式
- 模块布局规范和EDoc标注
- 用TCP/IP实现RPC服务
- 借助Telnet和服务器通信

什么！？没有“hello world”？

没错，没有“hello world”。我们已经在第2章概览了Erlang语言，现在该用它做些具体的事情了。为了简单快速地进入现实工作中的Erlang，我们对“hello world”说不！正相反，你所要实现的都是些能立即派上用场的东西。你要实现一个基于TCP的RPC服务器！

可能你还不知道这是个什么玩意儿，我们来解释一下。RPC表示远程过程调用（remote procedure call）。RPC服务器令你得以从远程机器上发起过程（也就是函数）调用。利用这个基于TCP的RPC服务器，人们只需要一个简单的TCP客户端（比如老式的Telnet）就能连上Erlang节点、执行Erlang命令，并检查执行结果。要想对上线投产后的软件进行诊断，这个TCP RPC服务器会是一个良好的起点。

源码

本书的源码托管于GitHub。访问http://github.com/并搜索“Erlang and OTP in Action”即可找到。你可以用git克隆整个代码库，也可以直接下载源码的zip压缩包。

将这个RPC应用部署到线上服务器上会造成一个安全漏洞，因为它允许用户在服务器上执行任意代码，不过问题并不大，只需对该工具可访问的模块和函数进行限制即可堵上这个漏洞。但在本章你不必在意这点。我们只是想借这个基本服务来阐述最基本、最强大，也最常用的一种OTP行为模式：通用服务器模式，即gen_server。（对行为模式一词，我们采用的是英式拼写behaviour，以期与Erlang/OTP文档保持一致。）对于构建于OTP行为模式之上的软件而言，OTP行为模式可以大大增强它们的稳定性、可读性和功能性。

在本章中，我们将带你一起实现你的第一个行为模式，你还将学习如何利用gen_tcp模块来进

行基本的TCP套接字开发（尽管从名称上看很像，但该模块并不是一个行为模式）。本书主要针对中级Erlang程序员，所以我们会直捣黄龙。你可得集中注意力，当然我们也会保证本章足够浅显易懂。待到学成之时，在可靠软件构建方面你将获得长足的进步。

到本章结束时，你的这个Erlang程序也将竣工，该程序最终会成为某产品级服务的一部分。在第4章中，借由OTP框架这个程序还会进一步成为Erlang应用，你可以用它和其他Erlang应用一起搭建完整的可交付投产的系统（也称为发布镜像）。再往后，到第11章时，你还会在随后开发的简易缓存应用中集成一个类似的服务。那将是一个更健壮、可伸缩性更强的TCP服务器，我们还将借此展现一些和该主题相关的有趣的窍门。现在，我们还是先来明确一下你在这一章所要完成的任务。

3.1 你所创建的是什么

该RPC服务器会监听TCP套接字并接受来自外来TCP客户端的连接。建立连接之后，客户端将可以通过TCP之上的简易ASCII文本协议执行函数调用。图3-1展示了该RPC服务器的设计和功能。

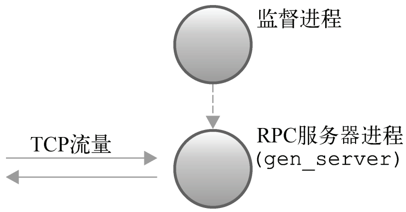


图3-1 RPC服务器进程通过套接字与外界相连。它通过TCP接收和执行请求，并将结果返回给客户端

该图展示了两个进程。一个是我们在第1章中提过的监督进程；由它派生出实际的RPC服务器进程。第二个进程会创建一个监听套接字等待其他人的连接。当连接到来时，它会从连接上读取描述普通Erlang函数调用的ASCII文本，并在执行调用后将结果通过TCP流返回。这类功能在很多场合都能派上用场，包括在紧要关头下的远程管理和诊断。此外，该RPC服务器遵循建立在TCP流之上的一个看似标准Erlang函数调用的基本文本协议。该协议的一般形式为：

```
Module:Function(Arg1, ..., ArgN).
```

例如：

```
lists:append("Hello", "Dolly").
```

（注意必须加上句号。）为了解释这些请求，RPC服务器会解析ASCII文本，提取出模块名、函数名和参数，并将它们转换成合法的Erlang项式。接着它会执行请求中的函数调用，并将调用结果所对应的Erlang项式格式化成ASCII文本，最终通过TCP流回传给客户端。

要完成这些工作，必须理解后续两小节所介绍的一些Erlang/OTP的基础概念。

3.1.1　基础知识提醒

你应该已经对模块、函数、消息和进程有了些基本的认识，这些概念我们在第1、2章就已经阐述过了。在介绍新的行为模式的概念之前，我们再来回顾一下这些概念。首先，在Erlang中，模块是函数的载体，进程借函数调用而诞生。进程彼此之间通过消息进行通信。图3-2展示了这些关系。

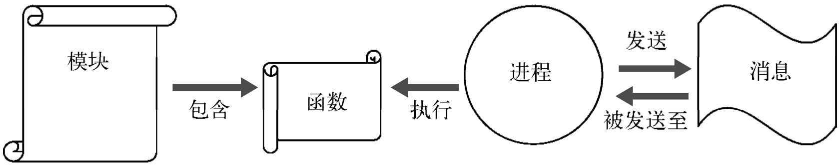


图3-2　模块、函数、进程和消息之间的关系

让我们再仔细回顾一下这些概念。

- 模块——模块是代码的容器。模块既可以将函数私有化，也可以将之导出供外部使用，模块通过这种方式得以控制函数的可访问性。每个目标文件（.beam文件）仅含一个模块。若模块名为test，则模块所在的源码文件名必须为test.erl，编译后的目标文件名也必须为test.beam。
- 函数——函数是真正干活的角色，Erlang模块中的所有代码都必须从属于某个函数。Erlang程序就是由一系列函数顺序拼接而成的。每个函数必须从属于某个模块。
- 进程——进程是Erlang并发的基本单元。它们相互之间通过消息进行通信。进程也是Erlang程序执行状态的基本容器：它可用于容纳会随时间变化的数据。每个进程都可以创建（派生）新的进程，并指定新进程应执行的函数调用。新进程将执行该调用并在调用结束时退出。被派生出来执行简单的`io:format/2`调用的进程会很短命，而被派生出来执行`timer:sleep(infinity)`这类调用的进程则永远也不会退出，除非外力强行将之终止。
- 消息——消息是进程交互的媒介。消息可以是任意的Erlang数据。进程间的消息传递是异步的，消息接收方总会拿到消息的一份单独的副本。消息送达后就被存放在接收方的信箱内，接收方进程可以通过`receive`表达式来获取消息。

快速理清了头绪，接下来便是行为模式的概念。

3.1.2　行为模式基础

行为模式是面向进程编程中各种常见模式的一种形式化表述。比如，服务器这个概念就非常通用，你所编写的进程之中很大一部分都符合这个概念。这些进程有很多共通之处——尤其是在遵循OTP监督规范等方面。每出现一种新的服务器进程就重新编写一遍这类代码是毫无意义的，这样做还会到处引入各种琐碎的bug和细微的差异。

相反，OTP行为模式将这类反复出现的模式分成了两个部分：通用部分和具体应用相关的实现部分。二者通过一套简单明确的接口进行通信。比如，在本章，你即将开发的模块也会包含这

么一个实现部分，它对应的正是OTP中最常见也最实用的行为模式：通用服务器，即gen_server。

1. 行为模式的组成部分

在日常交流中，行为模式这个词大有被滥用的嫌疑，它可指代下列多个概念：

- 行为模式接口；
- 行为模式实现；
- 行为模式容器。

行为模式的接口是一组特定的函数和相关的调用规范。gen_server行为模式的接口包含六个函数：init/1、handle_call/3、handle_cast/2、handle_info/2、terminate/2和code_change/3。

所谓实现，指的是由程序员提供的具体应用相关的代码。行为模式的实现是一个导出了接口所需的全部函数的回调模块。实现模块中还应包含一项属性-behaviour(...)，用以说明该模块所实现的行为模式的名称，这样编译器便可以协助检查模块是否完整地导出了接口所需的所有函数。代码清单3-1列出了模块的首部以及实现gen_server所必需的接口函数。

代码清单3-1　gen_server行为模式最精简的实现模块

```
-module(...).

-behaviour(gen_server).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {}).

init([]) ->
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

如果遗漏了其中的某些函数，该行为模式实现就无法完全满足gen_server接口的要求，这种情况下编译器会给出警告。等到下一节你开始挨个儿实现这些函数去创建你的RPC服务器的时候，我们再逐一解释每个函数的作用。

行为模式的第三个也是最后一个部分就是容器。容器是一个进程，它执行的是某个库模块中的代码，并且会调用与行为模式实现相对应的回调模块来处理应用相关的逻辑。（从技术角度说，容器也可以由多个密切相关的进程构成，但通常只有一个进程。）该库模块的名称与对应的行为

模式的名称一致。库模块中包含行为模式中的通用代码，其中也包括新容器的启动函数。例如，对于gen_server行为模式而言，这部分代码就位于Erlang/OTP库中stdlib部分的gen_server模块中。当你调用gen_server:start(...,foo,...)时，就会创建一个新的以foo为回调模块的gen_server容器。

在开发标准、并发、容错的OTP代码时，大部分的复杂性都可以靠行为模式容器来消解。库代码可以处理同步消息、进程初始化，以及进程清理与终止等事务，还提供了一系列挂接点，以便将模块融入代码变更框架[①]和监督树等更大的OTP模式和结构。

容器

容器这个词是我们自己选用的术语，不过我们觉得挺合适的。OTP 文档倾向于只谈论进程，而不去区分不同进程在行为模式中的职责，有时候会显得不清不楚。（如果你了解 Java 中的 J2EE 容器，就会发现其中有很多相似之处，但也有一些区别：OTP 容器很轻量，而且在 Erlang 行为模式中容器是唯一的实体。）

2. 行为模式的实例化

行为模式的目的在于为特定类型的进程提供一套模板。每个行为模式库中的模块都有一个或多个用于启动新的容器进程的API函数（通常名为start和/或start_link）。我们将新容器进程的启动称作行为模式的实例化。

进程类型

进程类型是一个通俗的说法（无论是否涉及行为模式），我们用它来指代诸如 gen_server 等同类进程。进程如果执行着大致相同的代码，就可以被归为同一类型，换言之它们所能理解的消息种类也大致相同。对于同一类型的两个进程而言，它们之间唯一的区别就在于各自的状态。同类型的进程通常也有着相同的派生签名（或初始调用），也就是说，它们具有相同的入口函数。

某些情况下，你会希望某个行为模式的实现模块在同一时间仅有一个实例；其他情况下，你又会希望创建出成千上万个跑着相同的代码却携带着不同数据的进程。重要的是，你得记住你的回调代码是在容器中执行的，而容器又是一个具有自己的身份和状态（包括信箱）的进程。这很像面向对象编程中的对象，只不过容器是并行存活着的代码执行者。

总结下来，行为模式接口就是让行为模式实现（你编写的代码）发挥出行为模式容器能力的契约。其目的在于简化遵循特定并发编程模式的进程的实现。使用OTP行为模式可以带来一系列好处：

- 开发者得以用更少的代码完成更多的事情——有时可以大大缩减代码量；
- 代码坚如磐石、稳定可靠，因为其核心库代码经过了严酷的测试；
- 代码可以被嵌入更大、提供更强劲功能的OTP框架，例如监督树；

---

① 代码变更框架用于处理代码热部署，即在不停止服务的情况下升级或降级服务代码，本书未详细讨论该主题。——译者注

- 代码更易于理解，因为遵循的是众所周知的模式。

对行为模式有了基本的了解之后，我们就要开始在上述内容的基础上实现RPC服务器了。从现在开始你所做的所有工作都与TCP RPC服务器的实现有关，这个练习涉及的东西很多。在某一层面上看，它讲述的是如何运用行为模式。你将开发出一套符合行为模式接口的行为模式实现，还会看到gen_server行为模式如何为你提供所需的各种功能。然而从另一层面上看，你在本章所做的事情还很基础：不过才刚刚开始在Erlang中使用OTP框架罢了。

3.2 实现 RPC 服务器

对于中等水准的Erlang程序员而言，模块、进程、函数以及消息传递等概念都不陌生。但他们的经验多半源自一些非正式的简单Erlang应用环境。在本章，我们将在OTP环境下重新审视这些概念。如果你还是个打算用本书来入门的Erlang新手，那么你很可能已经是一名有着不同的技术背景的程序员老手。这样的话，无须补充任何预备知识你也能看得懂本章的内容。

依我们之见，在*不使用*OTP的情况下直接用进程和消息传递编写纯Erlang代码（且做到万无一失）反而是更为高级的话题，只是万不得已而为之的下策。没有纯Erlang开发的经验或许也算是种福气，这样一来你便可以直接进入正规OTP开发方式的学习——甚至还包括一系列更为严格的编程规范，我们用这些规范来对模块的结构和布局、内联文档以及注释等方面进行约束。

鉴于你即将开发的行为模式实现必须以模块为载体，我们先从模块的创建和布局讲起。

3.2.1 行为模式实现模块的典型布局

行为模式的一大优点就在于它们的高度一致性。查看行为模式实现模块时，你一眼就能识别出这类模块中诸如行为模式接口函数这样的公共部分，以及start或start_link函数等自定义的部分。采用下述的行为模式实现模块的典型布局还可以进一步增加这些文件的可辨识性。

这份标准布局可分为4段。按照在源码文件中出现的顺序，表3-1分别列出了这些段落的详情：

表3-1 标准行为模式实现模块中的源码段落

段　落	描　述	有无导出函数	EDoc标注
首部	模块属性和样板内容	N/A	有，文件级别
API	编程接口；描述外界如何与模块交互	有	有，函数级别
行为模式接口	行为模式接口所需的回调函数	有	可选
内部函数	API和行为模式接口函数的辅助函数	无	可选

我们将从模块首部开始，依次考察这些段落的实现细节。

3.2.2 模块首部

在创建模块的首部之前，你得先建立一个用于容纳它的文件。既然是在开发基于TCP的RPC服务器，不妨将源文件命名为tr_server.erl。打开你最爱的编辑器，开工吧。

模块命名规范和扁平的命名空间

Erlang模块的命名空间是扁平的。换言之模块名可能会冲突。（Erlang也有一套类似于Java的实验性的软件包系统，但尚未广泛使用，功能支持也还不完整。）如果都用 server 之类的名字给模块命名，就很可能会导致不同工程中的模块重名。为了避免这类冲突，标准的做法是给模块名加上一个恰当的前缀。此处我们用的是 TCP 和 RPC 两个缩写的首字母，因此是：tr_server。

第一件事是录入位于文件首部的文件级注释：

```
%%%---------------------------------------------------------------------
%%% @author Martin & Eric <erlware-dev@googlegroups.com>
%%%  [http://www.erlware.org]
%%% @copyright 2008-2010 Erlware
%%% @doc RPC over TCP server. This module defines a server process that
%%%      listens for incoming TCP connections and allows the user to
%%%      execute RPC commands via that TCP stream.
%%% @end
%%%---------------------------------------------------------------------
```

注意，虽然注释用一个%就够了，但这里的每行注释都以3个%开头。这是文件级注释的规范，这些注释描述的是关于整个文件的信息。另外，这可能也是你头一回见到注释中包括EDoc标注。EDoc可以直接通过源码标注生成文档（与Javadoc类似），是随标准Erlang/OTP发行版一同发布的文档生成工具。限于篇幅，本书不打算深入讨论EDoc的使用：详情参见OTP文档中有关工具的章节。在此我们稍微多说两句，毕竟它是Erlang源码内联文档的事实标准。我们建议你玩转EDoc，并把它用到你自己的代码里。

EDoc标签都以@字符开头。表3-2描述了上述文件首部中出现的各个标签。等到第4章，解释完OTP应用的工作原理之后，我们还会继续讨论这个话题，届时我们会简要介绍如何运行EDoc生成文档。

表3-2　基本EDoc标签

标　　签	描　　述
`@author`	作者信息和邮箱地址
`@copyright`	日期和版权归属
`@doc`	常规文档文本。第一句是概要描述，其中可以包含XHTML和一些wiki标记
`@end`	标志着上述任意一种标签的完结。此处可以防止`%%%----...`这一行被误识别为前面的@doc标签的内容

除去注释，文件中的第一项就是`-module(...)`属性。此处的名称必须与文件名对应，在这里就是

```
-module(tr_server).
```

（请记住所有的属性和函数定义都必须以句号结尾。）紧跟在模块属性后面的就是行为模式属性。

它告知编译器这个模块是某行为模式的一个实现，如果你忘了实现并导出行为模式的某个接口函数，编译器便会给出警告。你所要实现的是一个通用服务器，所以应该添加如下的行为模式属性：

```
-behaviour(gen_server).
```

接下来是导出声明。通常要写两个（编译器会将它们合而为一，分开来是为了提高可读性）。第一个是你自己的API，第二个是行为模式接口要求导出的函数。由于你的API还没设计出来，放一个空的导出声明占位即可。但行为模式的接口函数都是已知的，罗列如下：


```
%% API
-export([]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
```

注意第二个声明上方的注释。行为模式的接口函数常被称作*回调*（callback）。这是因为容器启动时，行为模式实现模块的模块名会被传给新建的容器，而容器随后又会通过接口函数反向调用实现模块。我们将在本章逐一解释每个接口函数的用途。

紧随导出声明之后的是一系列可选的应用相关的声明和/或预处理定义。代码清单3-2完整地展示了tr_server模块的首部，并突出显示了这些声明和定义。

代码清单3-2　tr_server.erl的完整首部

```
%%%-------------------------------------------------------------------
%%% @author Martin & Eric <erlware-dev@googlegroups.com>
%%%  [http://www.erlware.org]
%%% @copyright 2008 Erlware
%%% @doc RPC over TCP server. This module defines a server process that
%%%      listens for incoming TCP connections and allows the user to
%%%      execute RPC commands via that TCP stream.
%%% @end
%%%-------------------------------------------------------------------
-module(tr_server).

-behaviour(gen_server).

%% API
-export([
         start_link/1,
         start_link/0,
         get_count/0,
         stop/0
         ]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).                         <-  ❶ 将SERVER设置为模块名
-define(DEFAULT_PORT, 1055).                      <-  ❷ 定义默认端口

-record(state, {port, lsock, request_count = 0}). <-  ❸ 用于保存进程状态
```

宏常被用于定义常量，这样一来要修改常量时只需修改一处代码即可（参见2.12节）。此处用宏定义了默认端口❷，并将SERVER定义成了模块名❶（服务器名后续还可能会修改，因此不要认为服务器名总是与模块名保持一致）。定义完宏，你还定义了一个记录，指定了该记录的名字和格式（参见2.11节），这个记录将被用于保存服务器进程的运行时状态❸。

首部已经介绍完了，我们来看一下行为模式实现模块的下一个段落：API。

3.2.3　API段

模块的所有功能都是通过应用编程接口（API）提供给用户的（用户才不会关心你的实现细节）。对于通用服务器而言，用户主要完成以下两件事：

- 启动服务器进程；
- 向进程发消息（并获取应答）。

gen_server提供了3个主要的库函数来实现这些基本功能。如表3-3所示。

表3-3　**gen_server**实现API的库函数

库　函　数	对应的回调函数	描　　述
gen_server:start_link/4	Module:init/1	启动并链接一个gen_server容器进程
gen_server:call/2	Module:handle_call/3	向gen_server进程发送同步消息并等待应答
gen_server:cast/2	Module:handle_cast/2	向gen_server进程发送异步消息

基本上，API函数只对这些库函数做了一些简单包装，以便在用户面前屏蔽实现细节。展现这些函数的工作机理的最佳手段，就是用它们来实现tr_server模块的API，如代码清单3-3所示。

代码清单3-3　tr_server.erl的API段

```
%%%===================================================================
%%% API                                                          <--  段落起始处的标题
%%%===================================================================

%%--------------------------------------------------------------------
%% @doc Starts the server.
%%
%% @spec start_link(Port::integer()) -> {ok, Pid}
%% where
%%  Pid = pid()
%% @end
%%--------------------------------------------------------------------
start_link(Port) ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [Port], []).   <--  派生服务器进程

%% @spec start_link() -> {ok, Pid}
%% @doc Calls `start_link(Port)' using the default port.
start_link() ->
    start_link(?DEFAULT_PORT).
%%--------------------------------------------------------------------
%% @doc Fetches the number of requests made to this server.
%% @spec get_count() -> {ok, Count}
```

```
%% where
%%   Count = integer()
%% @end
%%-----------------------------------------------------------------------
get_count() ->
    gen_server:call(?SERVER, get_count).              ❶ 调用方会等待应答

%%-----------------------------------------------------------------------
%% @doc Stops the server.
%% @spec stop() -> ok
%% @end
%%-----------------------------------------------------------------------
stop() ->
    gen_server:cast(?SERVER, stop).                   ❷ 无须坐等应答
```

请求❶使用的是`gen_server:call/2`，发送请求后调用方必须坐等应答。而`stop`❷这类简单命令通常会选用异步的`gen_server:cast/2`来实现。

一个模块一种进程

尽管我们可以同时从单个模块中派生出多个进程，但应该保证每个模块中的代码仅涉及一种类型的进程（由客户端执行的 API 代码除外，客户端可能是任意类型的进程）。如果模块的各个部分分别由不同类型的进程来执行，模块在系统中的角色便会不清晰，从而导致模块本身和整个系统变得难以理解。

从代码清单3-3列出的API中你可以大致看出tr_server所能做的3件事：

- 用`start_link()`或`start_link(Port)`启动服务器；
- 用`get_count()`查询服务器已处理的请求数；
- 调用`stop()`终止服务器。

在深入探讨这些函数的运作方式，以及调用者和服务器进程（容器）之间的通信细节之前，我们先来复习一下Erlang的消息机制。

1. 进程和通信的简要回顾

进程是所有并发程序的基石。它们通过投递异步消息相互通信，消息抵达之后便被放入接收进程的信箱中排队等候处理。图3-3展示了消息进入信箱的过程，在接收进程前来分拣消息之前，消息会被一直存放在这儿。

进程的这种自动缓存外来消息，并根据当前情况对消息进行选择性处理的能力是Erlang的一个关键特性（有关选择性接收的详情请参见2.13.2节）。

理解了上述原理，现在我们来看看OTP库是如何化解客户端与服务器之间消息传递的大量琐碎细节，从而给你提供出一整套更高层的进程通信工具的。 也许在灵活性上这些工具无法与你自行打造的通信模式相提并论，但它们简单、可靠，并且足以应对大部分的日常需求。同时它们还在超时、监督，以及错误处理等方面提供了保障。即便不使用OTP库，你也必须自行解决这些方面的问题（尽是些既枯燥又庞杂，还很难确保万无一失的工作）。

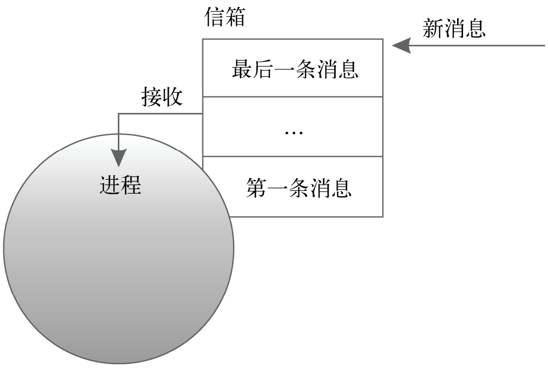


图3-3　消息抵达目的地之后，除非进程主动将消息取走，否则消息会一直停留在进程的信箱内。信箱的大小是没有上限的

2. 屏蔽协议

我们将进程所能接受的消息的集合称作该进程的协议。然而消息的这些细节是不应该暴露给用户的，因此API的主要职责之一就是对外界屏蔽协议。

你的tr_server进程可以接受下列简单消息：

- get_count；
- stop。

虽然二者都只是简单的原子，但tr_server模块的用户却没有必要了解这些实现细节，这些都应该屏蔽在API函数身后。不妨假想一个未来将会加入你的服务器的扩展功能，该功能要求用户必须先登录才能发送请求。于是你的API就得增加一个用于在服务器端创建用户的函数，如下所示：

```
add_user(Name, Password, Permissions) ->
  gen_server:call(?SERVER, {add_user, [{name, Name},
                                       {passwd, Password},
                                       {perms, Permissions}]}).
```

这类复杂消息格式不应该流于模块之外。如果客户端依赖于该格式，日后修改起来可就难了。将与服务器进行通信的逻辑封装于API函数内，模块的用户便可无视这些消息的格式。

最后，Erlang中的所有消息在底层都是异步发送的（通过!运算符），实际应用中，发送方在收到期望的应答之前其实什么也做不了。gen_server:call/2函数实现了可靠的同步请求–响应功能，其中默认应答等待超时为5秒，一旦超时便放弃等待应答（另外一个版本gen_server:call/3允许以毫秒为单位设置超时，同时也可以通过将超时设置为infinity来禁用超时）。

> **双重屏蔽**
>
> 此处还有一层屏蔽：OTP 库向你屏蔽了实际往返于进程之间的消息的细节。实际上你传给 call/2 和 cast/2 的消息数据参数仅仅是实际消息的载荷。其外围还会自动包上一层元数据，用于辅助 gen_server 容器区分消息类型（这样容器才知道该用哪个回调处理消息载荷），从而将应答回传给正确的进程。

现在，我们已经清楚地解释了API函数的目的以及用于实现它们的gen_server库函数，该回过神来看看代码了。

3. 深入API函数

代码清单3-3列出了各API的实现代码，现在我们来详细解释它们的作用。首先，表3-4总结了这4个API函数。

单例进程

简单起见，我们将这个服务器设计成了单例：同一时刻你只能运行该服务的一个实例。启动后，新的 gen_server 容器进程将以代码清单 3-2 中定义的 SERVER 宏为名进行注册（代码清单 3-3 中的{local, ?SERVER}参数的作用便在此）。这样一来 get_count()和 stop()才能够通过进程名与服务器通信。如果要同时启动多个服务器实例，还要对程序稍作修改，因为进程的注册名是不能重复的（参见 2.13.3 节进程注册表相关的内容）。不过，单例服务也是一种常用的建立系统级服务的手法，这些服务在每个 Erlang 节点上只能有一个实例（甚至整个 Erlang 集群中也只能有一个实例）。因此，单例服务器并不是那么不切实际的例子。

表3-4 **tr_server** API

函 数	描 述
start_link/1	启动tr_server并监听指定端口
start_link/0	同start_link/1，但使用默认端口
get_count/0	返回目前为止处理完成的请求数
stop/0	关闭服务器

这批API函数的工作机制如下。

- start_link(Port)和start_link()——启动并链接服务器进程。这是通过调用gen_server:start_link/4实现的，行为模式容器也正是通过该调用得知行为模式实现位于哪个回调模块中的（见第二个参数）。一般只需传入内置宏MODULE的值即可，该宏展开后是当前的模块名（参见2.12.1节）。

  ```
  gen_server:start_link({local, ?SERVER}, ?MODULE, [Port], [])
  ```

  执行这个调用时，会派生出一个新的gen_server容器进程，新进程将以SERVER宏展开后对应的名称在本地节点上注册，然后等待行为模式实现模块中的init/1回调函数完成进程初始化，最后返回（更多相关内容参见3.2.4节）。至此，服务器启动完毕并完成了所有初始化工作，已经准备好接受消息了。

  第三个参数，即此处的[Port]，为服务器提供了启动用的初始数据。该参数会被传给init/1回调函数，用于设置进程初始状态。第四个参数是附加参数列表，此处留空即可。注意，从API用户的角度来看，这些细节全都被屏蔽了，用户只须关心一个参数：即服务器的监听端口。

- get_count()——用gen_server:call/2向服务器同步发送原子get_count。也就是说该调用会等待服务器应答，从而临时挂起调用进程：

```
gen_server:call(?SERVER, get_count)
```

调用中的第一个参数必须是服务器进程的注册名或进程ID，此处用的是start_link/1函数中注册进程时所用的名称（SERVER宏）。调用中的第二个参数是要发送的消息。服务器收到消息并处理完毕后，会将应答回传给发起请求的进程。gen_server:call/2函数会负责接收应答并将之作为函数调用的结果返回，因此调用方完全不用关心消息收发。可以看到消息中的原子get_count（服务器协议的一部分）与相应的API函数同名。这样做有助于代码的阅读和调试——千万不要随心所欲地把服务器内部协议弄得晦涩不堪。

- stop()——用gen_server:cast/2异步发送原子stop（也就是说函数可以立即返回，无须等待应答）：

```
gen_server:cast(?SERVER, stop)
```

消息发送完毕后，你可以认为服务器一收到消息便会自行关闭。此处无须应答，所以选用cast而非call。

对于这个简单服务器而言，这些函数就够了。毕竟大部分功能都是经由TCP连接完成的，这几个API函数仅用于服务启停和状态检查。

4. @spec标签

在定义行为模式的接口回调函数之前，我们打算先简要介绍一下位于各函数上方文档中的EDoc标签（见代码清单3-3）。强烈建议你给每个API函数及整个模块都至少加上一个@doc标注（见代码清单3-2）。附加的@spec标签可用于描述函数输入参数的类型和返回值的类型。例如，start_link/1的@spec。

```
%% @spec start_link(Port::integer()) -> {ok, Pid}
%% where
%%  Pid = pid()
```

说明该函数可以接受一个整型参数并返回元组{ok, Pid}，其中Pid是一个进程标识符。类型名总是形如函数调用，因此不会与原子相混淆，比如integer()。你可以以::为分隔符将类型直接标注在Port等变量之后，也可以以where Pid = ...的形式将类型列在规格说明的末尾。

用户API部分就此告一段落，终于要开始实现行为模式的接口函数了——大部分实际功能都是由这些回调函数完成的。

3.2.4　回调函数段

你在API中用到的每个gen_server库函数都有一个gen_server行为模式接口指定的回调函数与之对应。现在该实现这些回调了。为了回顾相关内容，表3-5复述了表3-3，但额外加上了handle_info/2，API中用到的各个库函数无一与之对应。

首先回顾一下代码清单3-3中的tr_server:start_link/1函数，该函数对用户屏蔽了gen_server:start_link/4调用；由表3-5可以看出，新容器进程随后会回调tr_server:init/1

（根据gen_server行为模式接口的约定，tr_server模块必须导出该函数）来执行初始化操作。与此类似，tr_server:get_count/0对用户屏蔽了通信协议以及gen_server:call/2实现的通信细节。容器收到消息后，将回调tr_server:handle_call/2来处理消息。在这个例子中，这类消息只可能有原子get_count一种。同样，tr_server:stop/0用gen_server:cast/2向容器异步派发消息，收到这类消息后，容器会回调tr_server:handle_cast/2。

表3-5 **gen_server**的库函数及回调

库 函 数	对应的回调函数	描 述
gen_server:start_link/4	Module:init/1	启动并链接一个gen_server容器进程
gen_server:call/2	Module:handle_call/3	向gen_server进程发送同步消息并等待应答
gen_server:cast/2	Module:handle_cast/2	向gen_server进程发送异步消息
N/A	Module:handle_info/2	处理通过call或cast函数以外的手段发送给gen_server容器的消息。这些都是带外(out-of-band）消息

但请注意handle_info/2没有对应的gen_server库函数。这个回调是一个重要的特例，所有未经call或cast库函数发送至gen_server信箱的消息都由它处理（通常是直接用!运算符发送的裸消息）。出于多种原因，gen_server容器的信箱中会出现这类消息——例如，回调代码有可能向第三方请求数据。就你的RPC服务器而言，它接收的TCP数据离开套接字之后便会被转换为普通消息发送给服务器进程。

讲完了这些，代码清单3-4列出的正是你要实现的回调函数。

代码清单3-4 tr_server的gen_server回调

```
%%%===================================================================
%%% gen_server callbacks
%%%===================================================================

init([Port]) ->
    {ok, LSock} = gen_tcp:listen(Port, [{active, true}]),     ❶ 初始化服务器
    {ok, #state{port = Port, lsock = LSock}, 0}.

handle_call(get_count, _From, State) ->                        ❷ 返回请求数
    {reply, {ok, State#state.request_count}, State}.

handle_cast(stop, State) ->                                    ❸ 关闭gen_server
    {stop, normal, State}.
```

初始化服务器时，init函数会创建一个TCP监听套接字，设置初始的状态记录，还会立即触发一次超时。然后，是用于告知客户端进程当前已处理请求数的代码。在最后一个函数中，返回值stop比较特殊，用于让gen_server进程退出。

正如代码清单3-4所示，前3个回调函数都很简单。（我们把handle_info/2和另两个函数留到后面的代码清单3-5中讲解。）这3个函数中最复杂的部分就是它们的返回值的格式了，这些返回值被用于与gen_server容器进程通信。待我们来逐一进行分析。

- init/1，*初始化回调*——每次启动新的gen_server容器进程（比如通过gen_server:

start_link/4）这个函数都会被调用。这是OTP协助你以最小代价编写工业强度代码的第一个范例。start_link库函数会帮你完成相关设置，从而使你的代码能够顺利挂接到OTP强大的进程监督结构上。此外，它还提供了严格的进程初始化支持，在进程完成启动、注册（如果要求注册的话）和init/1回调调用之前，它会一直阻塞调用进程，从而确保你的进程在开始处理请求前完全准备就绪。

逐行来看这个函数，首先印入眼帘的是init([Port])->，其含义是init只接受一个参数，该参数是个仅含一个元素的列表，元素名为Port。请注意这和你在代码清单3-3中传给start_link/1函数的参数是一样的（即便只有一个元素，也请以列表的形式传入，这是init/1的一个通用规范）。

接下来，用标准库中的gen_tcp模块在指定的端口❶上建立一个TCP监听套接字：

```
{ok, LSock} = gen_tcp:listen(Port, [{active, true}]),
```

监听套接字是用于等待并接受外来TCP连接而建立的套接字。连接一旦建立，你便会得到一个可用于接收TCP报文的主动套接字。此处用到了{active, true}选项，其作用在于让gen_tcp把收到的所有TCP数据都以消息的形式直接发送给进程。

最后，init/1返回一个三元组，其中包含原子ok、初始进程状态（以#state{}记录的形式），以及一个莫名其妙的0：

```
{ok, #state{port = Port, lsock = LSock}, 0}.
```

这个0表示超时值。将超时置为零就是让gen_server容器在init/1结束后立即触发一次超时，从而迫使进程在完成初始化之后第一时间处理超时消息（由handle_info/2完成）。个中缘由我们稍后再作解释。

- handle_call/3，同步请求回调——每次收到由gen_server:call/2发送的消息时这个函数就会被调用。它接受3个参数：消息（跟传给call的一样）、From（暂且不去管它），以及服务器的当前状态（通过init/1设置，你想设置成什么都可以）。

  此处你只需处理一种同步消息：get_count。你所要做的就是从状态记录中提取出当前的请求数并将之返回。和上面一样，返回值是一个三元组❷，但内容与init/1稍有不同：

  ```
  {reply, {ok, State#state.request_count}, State}.
  ```

  它告诉gen_server容器你打算给调用方一个应答（理应如此，协议就是这样设计的）；回传给调用方的值应该是元组{ok, N}，其中N为当前的请求数；最后服务器的新状态应与原状态保持一致（此处未做任何改变）。

- handle_cast/2，异步消息回调——API函数stop()会使用gen_server:cast/2向服务器派发异步消息stop，发送完毕后无须等待任何应答。你的任务就是让服务器在收到消息后自行退出。通过cast发送的所有消息都由tr_server:handle_cast/2回调函数处理。这跟handle_call/3类似，只是此处没有From参数。当handle_cast函数看到stop消息时，只须返回下面的三元组即可：

  ```
  {stop, normal, State}.
  ```

这会告诉gen_server容器该停下来了❸（即进程终止），终止的原因是normal，也就是说这是一次正常退出。当前状态被原样传回（虽然后续也没机会再用它了）。注意随这个元组返回的原子stop用于让进程关闭，而API和服务器之间所使用的协议中的stop消息则可以换用任意原子（比如quit），我们只是选用了一个跟API函数stop()相同的名字而已。

至此，我们已经涵盖了gen_server行为模式相关的所有重点内容：接口、回调函数、容器，以及它们之间的交互。gen_server相关的内容肯定不止这些，这些内容将贯穿全书。至于你在此实现的服务器，还剩一个问题没有讨论：带外消息处理。很多服务器无须处理这类消息，但在这个应用中，所有的苦力活都是在这儿完成的。

1. 带外消息处理

我们曾经解释过，采用call或cast以外的手段发送给gen_server进程的所有消息都由handle_info/2回调函数处理。这些消息都被归类为带外消息，当你的服务器需要与第三方模块通信，而第三方模块又依赖于直接消息通信而非OTP库调用（比如套接字或端口驱动）时，它就派得上用场了。不过如果可能的话，还是应该尽量避免使用带外消息。

在init/1函数中，你为服务器建立了一个TCP监听套接字，还从该函数中返回了一个神秘的0超时值，从而立即触发超时（参见代码清单3-4）。

gen_server 超时事件

gen_server 设置了超时之后，一旦超时触发，就会产生一条由原子 timeout 构成的带外消息，这条消息将由 handle_info/2 回调处理。该机制常用于处理服务器在超时时间内未收到任何请求的情况，此时可以用它来唤醒服务器并执行一些指定操作。

这里有那么点儿滥用了超时机制的意思（但这也是个众所周知的技巧），其目的是让init/1函数尽快结束以免start_link(...)挂起。与此同时，你也能确保服务器立即跳到指定的代码段（handle_info/2的timeout子句）继续执行启动过程中更为耗时的部分——在这个例子中是等待你创建的监听套接字上的连接（参见代码清单3-5）。在这个应用中只有这里会用到超时，因此你可以确定自己绝不会再回到这里。

回到TCP套接字的问题上来：主动套接字会以消息的形式将收到的所有数据都转发给建立套接字的进程。（如果是被动套接字，你就必须不断询问还有没有数据。）你所要做的就是处理那些被转发过来的消息。对于gen_server而言，这些数据都属于带外数据，因此由handle_info/2回调函数处理，如代码清单3-5所示。

代码清单3-5　handle_info/2、terminate/2和code_change/3回调函数

```
handle_info({tcp, Socket, RawData}, State) ->      完成RPC请求后
    do_rpc(Socket, RawData),
    RequestCount = State#state.request_count,
    {noreply, State#state{request_count = RequestCount + 1}};    <—
handle_info(timeout, #state{lsock = LSock} = State) ->
    {ok, _Sock} = gen_tcp:accept(LSock),
    {noreply, State}.                                                递增计数
```

```
terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

这两个函数必不可少，但当前我们对它们不感兴趣

正如分析之前的3个回调一样，让我们来仔细看看这个函数。handle_info/2是带外消息回调。这个函数有两个子句，一个处理收到的TCP数据，另一个处理超时。超时子句很简单，前面已经作了解释，这也是服务器执行完init/1函数后所要做的第一件事（因为init/1将服务器超时置为零）：这实际上是一种延迟的初始化操作。这个子句调用gen_tcp:accept/1，在你建立的监听套接字上等待TCP连接（连接建立前服务器将一直阻塞于此）。连接一旦建立，超时子句便会返回，同时通知gen_server容器继续执行，进程状态则维持原样不变。（每个数据包都包含作为数据来源的套接字，因此你无须记录accept返回的套接字句柄。）

最后就是处理TCP消息的子句了，这些消息应当与模式{tcp, Socket, RawData}相匹配。主动套接字从TCP缓冲区中读取数据后发送给套接字所有者进程的正是这些消息。RawData是你所关心的域，其中包含客户端发送给你的ASCII文本。（总算回归到这个程序的根本目的了：处理TCP上的RPC请求！）数据处理是由代码清单3-6中列出的辅助函数do_rpc/2内的一大段代码完成的。RPC执行完毕后，你只需更新服务器状态中的请求计数（有关记录中域的更新请参见2.11.3节）并将控制权交还给gen_server容器即可。

2. 内部函数

本章一路看到这儿也挺不容易的，如果你对do_rpc/2函数的实现不感兴趣，我们也不怪你。你可以直接跳到3.3节（不过还是要先录入代码清单3-6中的代码）去看看如何运行这个服务器以及如何跟它进行TCP通信。但如果你想了解如何处理输入、解析和执行元调用（meta-call），就请接着往下看吧。

代码清单3-6　内部函数

```
%%%===================================================================
%%% Internal functions
%%%===================================================================

do_rpc(Socket, RawData) ->
    try
        {M, F, A} = split_out_mfa(RawData),
        Result = apply(M, F, A),
        gen_tcp:send(Socket, io_lib:fwrite("~p~n", [Result]))
    catch
        _Class:Err ->
            gen_tcp:send(Socket, io_lib:fwrite("~p~n", [Err]))
    end.

split_out_mfa(RawData) ->
    MFA = re:replace(RawData, "\r\n$", "", [{return, list}]),
    {match, [M, F, A]} =
        re:run(MFA,
               "(.*):(.*)\s*\\((.*)\s*\\)\s*.\s*$",
               [{capture, [1,2,3], list}, ungreedy]),
```

❶ 执行请求中的调用
❷ 输出结果
❸ 丢弃回车换行符
❹ 解析请求串

```
        {list_to_atom(M), list_to_atom(F), args_to_terms(A)}.

args_to_terms(RawArgs) ->
    {ok, Toks, _Line} = erl_scan:string("[" ++ RawArgs ++ "]. ", 1),
    {ok, Args} = erl_parse:parse_term(Toks),
    Args.
```

纵观整个代码清单，代码清单3-6中的代码可分为4个主要部分：切分输入、解析函数参数、执行请求中的调用，以及回传结果。首先，可以看到do_rpc/2函数的内部逻辑被悉数包裹在try表达式内（参见2.8.2节）。这是因为你需要处理来自外部世界的数据，可能会出现各种错误，而try不失为一个简单易行的崩溃处理手段，在代码崩溃时（抛出异常）打印出错误信息然后继续执行，总好过整个服务器进程崩溃。另一方面，这种手段无法对付3.3节中即将提到的语法和语义都正确的恶意代码。

3

边界检查

当数据从不可信的外部世界进入可信的内部区域时应该对数据进行检查，这是 Erlang 程序设计的一个基本原则。倘若经过验证，数据符合你的预期，就没必要做重复检查了：写代码时只需关注正确情况即可，剩下的问题可以全权交由监督机制处理。这样做可以大大缩减代码的尺寸，编程错误的数量也会因可读性的提升而减少。至于其余的错误，由于你并不刻意掩饰，进程因崩溃而重启时它们自然会被日志记录在案，从而令你得以在问题出现的第一时间着手解决。让崩溃来得更猛烈些吧！

首先使用标准库中的re模块（Perl兼容的正则表达式）去除行末的回车换行❸。剩下的应该是形如Modul:Function(Arg1,...,ArgN)的格式化文本，该格式遵循的正是3.1节起始处所定义的协议。（否则，你的代码必将在某处崩溃，然后落入try表达式的崩溃处理逻辑。）

接着，再次使用re模块，分别提取出Module、Function和Arg1,...,ArgN❹。正则表达式的用法不在本书的讨论范畴内，详情请参照标准库文档。模块名和函数名都应遵循Erlang原子的命名规范，这样才能将它们从字符串转换成原子。

针对参数的处理就比较复杂了。参数部分是一个由逗号分隔的项式列表，其中可以包含零个、一个，或多个参数。参数处理由args_to_terms/1完成，此处用到了两个标准库函数，分别用于将字符串切分成语元，以及将语元解析成真正的Erlang项式列表。

I/O 列表：轻松实现分散写入/集中读取（scatter / gather）

io_lib:fwrite/2 的结果不一定是普通字符串（即扁平字符列表）。即便如此，你仍然可以直接将结果传给套接字，这个结果被称为 I/O 列表：它是一个可以深层嵌套的列表，既可以包含字符编码也可以包含二进制数据块。通过这种方式，在依次输出多个 I/O 列表时，就不用再为了拼接所有数据而专门创建一个中间列表了：你只需创建一个包含所有数据段的列表，然后将之整个传给输出流即可。这很类似于现代操作系统中的分散写入/集中读取技术[①]。

① 实际上这里只体现出了分散写入。——译者注

随后，将模块名、函数名、参数项式列表传给内置函数apply/3❶。该函数与spawn/3非常类似（参见2.13节），但它并不会启动新进程——它用于执行相应的函数调用（所以我们也称它为元调用运算符）。最后，该函数的返回值由io_lib:fwrite/2转换为格式化文本❷，用作回传给用户的响应，通过套接字发送回去——至此，远程过程调用执行完毕！

RPC服务器已经竣工，可以运行了。在下一节，你可以尝试运行，看看它能否正常运转。

3.3 运行 RPC 服务器

执行前的第一步就是编译代码。（如我们在本章开头处所说，你可以在GitHub.com上找到完整的源码。）执行命令erlc tr_server.erl。没有出现错误的话，当前目录下就会多出一个名为tr_server.beam的文件。在同一目录下启动Erlang shell，然后启动服务器：

```
Eshell V5.6.2  (abort with ^G)
1> tr_server:start_link(1055).
{ok,<0.33.0>}
```

此处我们选了一个很好记的端口号1055（10 = 5 + 5）。start_link调用返回了一个元组，包含原子ok和新服务器进程的进程标识符（不过当前你还用不上它）。

接下来，向1055端口发起一个Telnet会话。在大部分系统上，直接在系统shell提示符下（不是Erlang shell）输入telnet localhost 1055即可（不过，某些版本的Windows上没有Telnet——需要的话可以去下载一个免费的Telnet客户端，比如PuTTY）。例如：

```
$ telnet localhost 1055
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
init:stop().
ok
Connection closed by foreign host.
```

首次会话成功！为什么这么说？让我们来仔细分析一下整个会话过程，看看到底发生了些什么。

首先，你用Telnet通过TCP的1055端口连接至运行中的tr_server。连接成功后，你输入了文本init:stop().，服务器随即读取并解析这段文本。可以预料到服务器将会调用apply(init, stop, [])。同时你也知道init:stop/0会返回原子ok，这与你看到的打印结果相符。不过接下来你看到的却是“Connection closed by foreign host.”。这是Telnet打印的，因为突然间远程端连接的套接字被关闭了。这是因为init:stop()关闭了运行着RPC服务器的整个Erlang节点。这个例子不仅演示了RPC服务器的工作原理，还演示了让人不受限制地在你的节点上随意运行代码是多么危险！你可以在后续的改进版本中对用户做出限制，令用户只能调用特定的函数，你甚至还可以配置这些限制。

服务器不应该调用自身

通过 RPC 服务器，你可以调用服务器端任意模块中导出的任意函数，只有一个例外：你自己的 tr_server:get_count/0。一般来说，服务器不能调用自己的 API 函数。假设你在

某个回调中向同一个服务器发起一个同步调用：比如，试图在 handle_info/2 中调用 get_count/0API 函数。这种情况下，服务器向自己发起 gen_server:call(...)调用请求。而该请求只有在 handle_info/2 返回后才能得到处理，从而引发循环等待——服务器便会陷入死锁。

总结下来，没用多少行代码，你就构建出了一个切实可用的应用（不过还得略微做些调整）。更重要的是，这是个能够与OTP框架融合的稳定的应用。

3.4　浅谈测试

作为一名严谨的开发者，收工之前你还有一件事情要做：添加测试。不少人会争辩说从一开始就应该进行测试，你应该用测试来引导开发过程才对。但在示例中增加测试代码会干扰我们的视线，偏离我们的主旨。毕竟这本书讲的是OTP框架，而不是测试。而且，你在此开发的都是并发分布式系统，要给它们编写测试，其间的奥秘本身就足够写上一两本书了。

有两个层面的测试与开发者直接相关：单元测试和集成测试。单元测试关注的是编写一键运行的测试，这些测试针对的是程序中的某些特定属性（每个测试最好只覆盖其中一个属性）。集成测试则更注重于测试多个独立开发的组件之间的协同工作，并且往往需要先进行一些人工设置才能运行。

Erlang/OTP标准发布镜像中包含两个测试框架：EUnit和Common Test。EUnit主要用于单元测试，它的目的在于尽量简化开发过程中测试代码的编写和运行。Common Test基于所谓的OTP Test Server，是个更加重型的框架，通过它你可以一边在一台或多台机器上运行测试，一边将测试结果集中记录到运行着框架的机器上。它更适合于运行晚间例行集成测试等大规模测试。你可以在Erlang/OTP文档的工具一节找到关于这两个框架的详细介绍。

在此我们向你简要介绍一下如何使用EUnit编写测试用例，非常简单。首先，在-module(...)声明后面加入一行代码：

```
-include_lib("eunit/include/eunit.hrl").
```

最麻烦的部分便就此处理完毕了。接下来，找些东西来测一测吧！比如，你可以测试一下服务器能否成功启动。测试代码必须放在函数内，该函数必须没有任何参数且函数名要以_test结尾。EUnit探测出这些函数后会将它们统统当作测试。如果测试函数正常返回，便视作成功；如果抛出异常，便视作失败。因此，你的测试可以写成：

```
start_test() ->
    {ok, _} = tr_server:start_link(1055).
```

回想一下，=是匹配运算符，如果右侧的值无法匹配左侧的模式，该运算符将抛出badmatch错误。也就是说只有服务器成功启动该函数才能正常返回；对于任意其他情况，start_test()都会引发异常。就是这么简单！

要运行测试，必须先重新编译模块。然后，你可以在Erlang shell输入：

```
eunit:test(tr_server).
```

或者

```
tr_server:test().
```

二者效果相同：作用都是运行tr_server模块中的所有测试。请注意你并未编写过名为`test()`的函数：该函数是由EUnit自动生成的，同时EUnit会自动导出你编写的所有测试函数。

EUnit的很多特性可以帮助你写出极为紧凑的测试代码，其中包括一系列实用的宏，它们会随着你所包含的eunit.hrl头文件自动加载进来。我们建议你通过阅读Erlang/OTP文档中的EUnit用户手册来获取更多相关信息。

3.5　小结

我们在本章涵盖了大量内容，纵观了OTP行为模式的各种基础知识以及构成这些行为模式的3个组成部分：接口、容器和回调模块。我们还通过实际示例深入介绍了`gen_server`行为模式。

在下一章中，这个小巧独立的通用RPC服务器将被搭载进更大的结构之中，成为一个企业级OTP应用。迈出这一步之后，你的服务器便会演变成一个版本化的、容错的应用，既可以在其他项目中为他人所用，也可以直接上线投产。下一章将会讲授基本的容错结构（监督者）以及将各种功能打包成精良的OTP包的方法，这些内容将会让你对Erlang/OTP框架的基本认识更上一层楼。

第 4 章

OTP应用与监督机制

本章概要

- OTP应用简介
- 用OTP监督者实现容错
- 用EDoc生成文档

整个Erlang/OTP生态系统的目的就在于构建稳定、容错的系统。我们在第3章介绍了该生态系统中的核心概念，并搭建了一个简单的RPC服务器；现在我们将更进一步，教你如何合理地将这个应用打包成容错的、产品级质量的服务。为此，我们先要介绍两个新的基本概念。

- 应用是Erlang对相关模块进行打包的一种手段。打包的目的并不在于发布，而在于使这些模块成为一个整体。有一部分OTP应用仅仅是供他人调用的库代码，但大部分应用都具有自己的生命周期：启动，完成预设任务，最后关闭。部分应用可以同时运行多个实例，另一些应用则仅限一个。
- 监督者是OTP最为重要的一个特性。它们负责监控其他进程，并在出现问题时重启故障进程或向上汇报侦测到的问题。你还可以将监督者分层组织成监督树进而构建出高度可靠的容错系统。

在本章中，我们不打算深入介绍应用和监督者的理论及实践。我们将集中精力武装第3章中搭建的模块，把它包装成一个OTP应用并为它设置监督者。我们会讨论全章各项任务的大部分基本要点，并阐述每个步骤背后的动机。等到迈入本书的第2部分后，我们还将详细讨论用于高级监督功能和应用代码升级处理的各种有意思的选项，而且还会将多个应用进一步装配成更大的、称为发布镜像（release）的结构。

4.1 OTP 应用

首先我们来看看代码应该经由怎样的组织才能有效融入正常的Erlang/OTP系统。出于种种原因，这个问题困扰过很多新手。刚刚接触Erlang时，OTP简直就像是某种暗黑魔法，文档拙劣、示例奇缺。我们一路崎岖，不断试错，顺着Erlang邮件列表中的各位前辈的点滴指引艰难地学习着这套强大的系统。所幸在摸清了基本概念之后，事情还是相当简单明了的。

术语：应用

在 OTP 环境下，应用一词有着特殊的含义：应用就是由磁盘上的一系列模块和若干额外的元数据文件按一定规范组织起来形成的软件组件。通过这种组织方式，系统便可以知晓当前已经安装了哪些应用，同时也让你能够按应用名来启动或停止应用。

从现在起，除非特别说明，否则我们所说的应用都是 OTP 意义下的应用。

浅显地说，OTP应用无非就是一组相互关联的代码。我们将其中一部分称为*库应用*：这些应用纯粹是供其他应用调用的一系列模块的集合。（Erlang/OTP的stdlib就是库应用的一个实例。）还有一些应用则更为常见，它们具有自己的生存周期，启动之后会运行上一段时间，最后终止。我们将这类应用称为*主动应用*。每个主动应用都配有一个负责对应用进程进行管理的根监督者。我们将在4.2节详细介绍监督者。

4.1.1　OTP应用的组织形式

创建OTP应用时的主要工作集中于标准目录结构的建立和应用元数据的编写。元数据的作用在于让系统获悉应该如何启动和停止应用，还可用于指定应用的依赖项，也就是在应用启用前必须预先安装或启动哪些其他应用。创建主动应用时，还需要写一点儿代码，这部分内容将在4.1.3节详述。

主动应用与库应用

主动应用与库应用采用的是相同的目录结构和元数据文件，都可以融入 OTP 整体应用框架。二者之间的主要区别就在于主动应用具有一定的生命周期，必须先启动才能发挥作用。与之相反，库应用只是一组供其他应用调用的被动模块集，不涉及应用启停。由于本书的重点在于主动应用的开发，除非特殊说明，否则我们所提及的应用都是指主动应用。

如图4-1所示，Erlang/OTP应用的目录布局很简单。许多熟悉Erlang但却不了解OTP的人在开发应用时采用的也是这个目录结构，只是没有用上元数据。

```
<application-name>[-<version>]
                                |
                                |- doc
                                |- ebin
                                |- include
                                |- priv
                                |- src
```

图4-1　OTP应用的目录布局。目录名可以包含版本号。标准子目录包括doc、ebin、include、priv和src，其中只有ebin是必需的

其中<application-name>显然应该换成你自己的应用名，此处是tcp_rpc。[-<version>]是可选项：开发时用不到它，但在交付时通常会采用tcp_rpc-1.0.2这样的目录名，这样做可

以简化今后的代码升级工作。图4-1中列出的各个子目录的作用光看目录名便一目了然，不过我们还是在表4-1中给出了详细描述。

表4-1 应用目录下的子目录

目 录	描 述
doc	用于存放文档。如果文档是用EDoc生成的，请将overview.edoc文件放在此处，其余的文件将会自动生成
ebin	用于存放编译后的代码（.beam文件）。含有应用元数据的.app文件也应存放在此处
include	用于存放公共头文件。所有作为公共API的一部分的.hrl文件都应该放在这个目录中。仅用于你自己的代码之中且不打算公开的私有.hrl文件则应该与其余的源码文件一起放在src目录下
priv	用于存放各种需要随应用一起发布的其他内容。包括但不限于模板文件、共享对象文件和DLL等。定位应用priv目录的方法很简单：调用`code:priv_dir(<application-name>)`，便会以字符串形式得到priv目录的完整路径
src	用于存放应用的源码。不仅包括Erlang的.erl文件和内部.hrl文件，也包括ASN.1、YECC、MIB等其他源文件（如果不打算随应用一起发布源码，可以省去该目录或将目录留空。）

请立即按照这个结构建立好相应的目录，然后把第3章的源码放入src目录中。

4.1.2 为应用添加元数据

应用的目录结构已经建立完毕，可以添加OTP所需的元数据了。这些元数据以普通Erlang项式描述，位于ebin目录下的一个名为<application-name>.app的文本文件中。代码清单4-1展示了ebin/tcp_rpc.app文件，也就是tcp_rpc应用的元数据。

代码清单4-1 应用元数据文件：ebin/tcp_rpc.app

```
%% -*- mode: Erlang; fill-column: 75; comment-column: 50; -*-

{application, tcp_rpc,
 [{description, "RPC server for Erlang and OTP in action"},
  {vsn, "0.1.0"},
  {modules, [tr_app,
             tr_sup,
             tr_server]},
  {registered, [tr_sup]},
  {applications, [kernel, stdlib]},
  {mod, {tr_app, []}}
 ]}.
```

这个.app文件的作用在于告诉OTP如何启动应用，以及该应用应该如何与系统中的其他应用相融合。再重复一遍，我们的首要目的并不在于打包发布，而在于组装更大的可启动、停止、监督和升级的功能单元。

.app文件的格式很简明。除去注释，只剩下一个由句号结尾的Erlang项式：三元组`{application, ..., ...}.`。其中第二个元素是应用名称所对应的原子，此处即是`tcp_rpc`。第三个元素是一个参数列表，其中每个参数都是`{Key, Value}`对的形式，有些是必需的，有些是可选的。此处罗列的是大部分应用都会用到的最为重要的那些参数，在本书第二部分我们还将详述这部分内容。表4-2按代码清单4-1中的出现顺序分别描述了这些参数。

表4-2　.app文件中的主要参数

参　　数	描　　述
description	针对应用的简短描述。通常只有一两句话，但你想写多长都可以
vsn	应用的版本。版本可以用任意字符串表示，但我们建议你坚持采用标准的<主版本号>.<次版本号>.<修订版本号>格式：虽然Erlang/OTP本身并不关心你使用什么字符串来表示应用的版本，但某些程序会出于特定目的自行解析应用的版本字符串，胡乱选取版本字符串很可能会让这些程序功能错乱
modules	应用中的模块列表。这个清单的维护工作甚是乏味，不过有现成的工具可以助你一臂之力。模块在列表中的顺序无关紧要，但按字典序对模块进行排序会较为易于维护
registered	我们知道，对Erlang进程进行注册（2.13.3节）后， 便可以用注册名来定位进程。这种手法常用于系统服务等场合。在.app文件中罗列出所有进程注册名并不会促使系统执行实际的注册操作，但这可以告知OTP系统哪个进程注册了哪个名字，从而为系统升级等操作提供便利，同时也可以尽早发现重复的注册名并给出警告
application	必须在该应用启动前先行启动的所有应用。应用往往会依赖于其他应用。主动应用要求自己所依赖的所有应用在自己的生命周期开始之前先行启动并就绪。列表中各应用的顺序无关紧要——OTP很智能，它会纵观整个系统并明辨每个应用的启动时机
mod	告知OTP系统应该如何启动应用。该参数的值是一个元组，其内容为一个模块名以及一些可选的启动参数。（不要把通用配置信息写到这些参数中——请使用正规的配置文件来存放通用配置信息。）这个模块必须实现application行为模式，请参见4.1.3节

截至目前为止，你已经建立了应用的目录结构，元数据也已经就位。（如果尚未创建ebin/tcp_rpc.app文件，请先创建该文件。）但整个应用还远未完工。表4-2中mod的描述中曾提到，你的应用还需要一个启动入口，它必须是application行为模式的一个实现模块。下一小节将详述这个环节。

4.1.3　应用行为模式

每个主动应用都配有一个application行为模式的实现模块。该模块用于实现系统启动逻辑。它至少要负责根监督者的启动，该监督者将成为应用中其他所有进程的鼻祖。根据系统需要，应用行为模式模块还可以完成一些其他任务。我们将在4.2节详细阐述监督者。现在，让我们将注意力集中到src/tr_app.erl文件中应用行为模式的实现上，如代码清单4-2所示（去掉了注释）。

> **应用行为模式的命名**
>
> 应用行为模式的实现模块通常被命名为<application-name>_app。

代码清单4-2　应用行为模式：src/tr_app.erl

```
-module(tr_app).

-behaviour(application).        % 行为模式声明

-export([
     start/2,                   % 应用行为模式的
     stop/1                     % 回调函数
```

```
    ]).

start(_Type, _StartArgs) ->
    case tr_sup:start_link() of          <— 启动根
      {ok, Pid} ->                          监督者
          {ok, Pid};
      Other ->
          {error, Other}
    end.

stop(_State) ->
    ok.
```

这个小模块应该很容易理解，如果需要，你可以向前查阅第2章来复习行为模式的实现模块。在这里，你实现了一个application行为模式，该行为模式要求导出两个回调start/2和stop/1。（这个模块没有任何用户API，因此除了这两个回调以外这里别无其他导出函数。）

此处的stop/1回调很简单：应用关闭时无须做出任何特殊处理，忽略输入参数并直接返回ok即可。真正让人操心的是start/2。OTP系统会在应用即将启动时调用该函数，它负责完成实际的启动工作并以{ok, Pid}的形式返回根监督者的进程ID。其他各种需要在应用启动时完成的任务，如配置文件的读取、ETS表的初始化等，也可以在此进行。仅就这个简单的tcp_rpc应用而言，我们只需调用tr_sup:start_link()函数（尚未提及）来启动根监督者便可。（监督者模块tr_sup留待后续的4.2节实现。）紧接着你得检查start_link()的返回值，如果不对劲就立即上报一个错误。start/2的输入参数暂时可以忽略，不过为了满足你的好奇心，我们解释一下，此处的Type一般取值为normal，但也可能是{failover,...}或{takeover,...}，StartArgs则是你在.app文件中传给mod的参数。

4.1.4 应用结构小结

总结一下，建立OTP应用要做3件事：

(1) 遵循标准目录结构；

(2) 添加用于存放应用元数据的.app文件；

(3) 创建一个application行为模式实现模块，负责启动应用。

另外还有一个有待探讨的细节，就是应用行为模式中的start/2函数是如何启动根监督者的。主动应用的目的就在于启动一个或多个进程以完成特定的任务。为了加强控制，这些进程应该由监督者——也就是实现了supervisor行为模式的进程——统一派生和管理。

4.2 用监督者实现容错

监督者是Erlang/OTP的核心之一。主动OTP应用由一个或多个进程组成，它们相互协作共同完成任务。监督者间接启动这些进程，对这些进程负责，并在必要时重启它们。本质上说，在运行时，应用就是一棵由监督者和工作进程共同构成的进程树，树根就是根监督者。图4-2展示了一个可能的假想应用的进程结构。

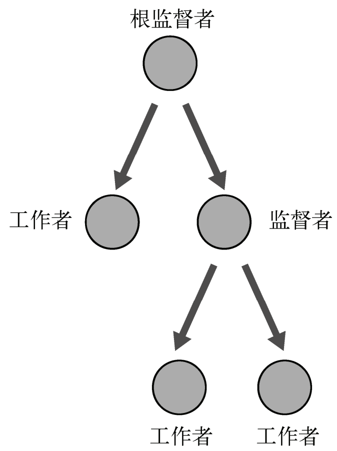


图4-2　某假想应用的进程树。这个例子包含一个根监督者、一个由根监督者直接派生的工作进程和一个子系统的监督者，该子系统本身又由另外两个工作进程组成

你可以通过编写`supervisor`行为模式的实现模块来创建监督者。如果工作进程本身就是基于OTP行为模式的（如`tr_server`），为其设置监督者将会很容易。`gen_server`、`gen_event`和`gen_fsm`等标准OTP工作者行为模式融入监督树的方式一点儿也不神奇——无非就是实现一些接口、遵循返回值方面的约定并设置好进程间的链接。幸运的是，你不必了解其中的详情。如果监督树中碰巧出现了未遵守标准行为模式的代码，你可以借助标准库中的supervisor_bridge适配器来处理相关问题。

根监督者行为模式模块的命名

根监督者行为模式的实现模块通常被命名为<application-name>_sup。

4.2.1　实现监督者

代码清单4-3展示的是src/tr_sup.erl，也就是tcp_rpc应用的根监督者的实现。相较于tr_app，该模块要稍微复杂一些，特别是除了行为模式接口的回调之外，这个模块还提供了一个API函数。通过它你才能在tr_app模块中启动监督者。（实际上，`tr_sup:start_link()`完全可以融入`tr_app:start/2`，不过我们倾向于将这部分职能隔离出来，而不是把监督者相关的逻辑混入_app模块。）

代码清单4-3　根监督者实现

```
-module(tr_sup).

-behaviour(supervisor).

%% API
-export([start_link/0]).

%% Supervisor callbacks
```

```
-export([init/1]).

-define(SERVER, ?MODULE).

start_link() ->                                                    ❶启动监督者
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

init([]) ->                                                        ❷指明如何启动和管理子进程
    Server = {tr_server, {tr_server, start_link, []},
              permanent, 2000, worker, [tr_server]},
    Children = [Server],                                           ❸指明监督者的行为
    RestartStrategy = {one_for_one, 0, 1},
    {ok, {RestartStrategy, Children}}.                             返回监督规范
```


start_link()API函数仅负责监督者的启动，具体来说就是以模块名为参数调用库函数supervisor:start_link/3❶。（与代码清单3-3中通过gen_server:start_link/4启动tr_server的方法类似。）其中第一个调用参数是二元组{local, ?SERVER}，用于让OTP库在本地节点上以tr_sup为注册名自动注册监督进程（SERVER的定义与MODULE相同）。第三个参数是传给回调函数init/1的启动参数。由于init/1无须任何输入参数，此处仅需传入一个空表。

真正有意思的内容都集中在init/1函数里。子进程的启动策略、管理策略❷以及监督者进程本身的行为❸都是经由该函数的返回值告知给OTP监督者库的。

监督者可以完成多种任务，本书的第二部分将对此进行细致的考察。现阶段你只需要理解这个案例中涉及的内容就可以了。我们不妨由浅入深：先从重启策略讲起。

4.2.2 监督者重启策略

init/1回调函数的返回值的格式为{ok, {RestartStrategy, Children}}，其中Children是若干子进程规范组成的一个列表，这些规范比较复杂，我们留待下一节详述。RestartStrategy则比较简单，它只是一个三元组{How, Max, Within}，此处它的值是：

```
RestartStrategy = {one_for_one, 0, 1}
```

这里的How取值为one_for_one，表示一旦有子进程退出，监督者将针对该进程，且仅针对该进程进行重启。该重启操作不会影响同时运行的其他进程，如图4-3所示。（本书后续内容中还会介绍其他策略。例如，某些策略会对子进程进行编组，组中任意一个进程的意外终止都将导致整个进程组的重启。）

Max和Within这两个值（此处分别取值为0和1）是相互关联的：它们共同确定了*重启频率*。第一个值指定的是最大重启次数，第二个值指定的是时间片。例如，当Max=10、Within=30时表示监督者最多可以在30秒内重启子进程10次。一旦超过限制，监督者就会在终止所有子进程后自我了断，并顺着监督树向上汇报故障信息。这些数值的取值很大程度上取决于应用本身，因此我们无法在此给出一个普适的推荐值，不过生产系统中经常会将该频率设置为每小时（3600秒）4次。此处这两个值分别取值为0和1表示目前不启用自动重启，因为自动重启会妨碍我们发现代

码中的潜在问题。错误日志和重启事件的检测将留待第7章讨论。

下面来看init/1返回值中的Children部分。这是一个元组列表，其中每个元组表示一个受监督的子进程。在这个案例中，子进程只有一个，即你的服务器。该列表长度不限，同时管理半打子进程的监督者也不少见。监督者也可以在启动后动态增删子进程，但大多数情况下这种静态列表就足以满足需要了。

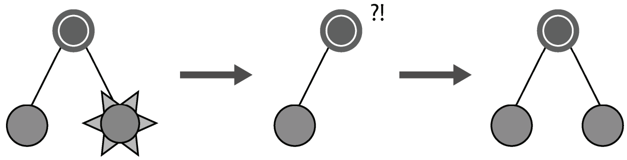


图4-3　一对一重启策略：监督者在需要重启子进程时将各个子进程看作相互独立的个体。崩溃的子进程不会影响正常的子进程

4.2.3　编写子进程规范

子进程规范是一个用于描述受监督者管理的进程的元组。对于大多数监督者而言，子进程会随监督者的启动而启动并在监督者的生命周期结束时退出。对于单个需要监督的进程，init/1函数给出的描述如下：

```
Server = {tr_server, {tr_server, start_link, []},
          permanent, 2000, worker, [tr_server]}
```

子进程规范由6个元素组成：{ID, Start, Restart, Shutdown, Type, Modules}。

- 第一个元素ID，是一个用于在系统内部标识各规范的项式。简单起见，此处采用的是模块名，即原子tr_server。
- 第二项Start，是一个用于启动进程的三元组{Module, Function, Arguments}。与调用内置函数spawn/3时一样，其中第一个元素是模块名，第二个元素是函数名，第三个元素是函数的调用参数列表。在这个例子中，监督者应调用tr_server:start_link()来启动子进程（也就是tr_server）。
- 第三个元素Restart，用于指明子进程发生故障时是否需要重启。此处指定为permanent，因为你搭建的是需要长期运行的服务，无论出于任何原因导致进程终止都应重启进程。（该选项还可取值为表示永不重启进程的temporary，以及仅在进程意外终止时重启进程的transient。）
- 第四个元素Shutdown，用于指明如何终止进程。此处取值为一个整数（2000），表示终止进程时应采用软关闭策略，给进程留出一段自我了断的时间（以毫秒为单位），如果进程未能在指定时间内自行退出，将被无条件终止。该选项还可取值为brutal_kill，表示在关闭监督进程时立即终止子进程；以及infinity，主要用于子进程本身也同为监督者的情况，表示应给予子进程充分的时间自行退出。

- 第五个值Type，用于表示进程是监督者（supervisor）还是工作者（worker）。在整个监督树中，除了实现了supervisor行为模式的监督者进程以外，剩下的都是工作进程。随着应用的复杂度的提升，你可以按自己的喜好组织监督进程进而形成层级结构，以提供更细粒度的控制（参见图4-2）。监督者可以通过Type字段识别子进程是否同为监督者。显然此处的服务器进程是工作进程。
- 第六个选项列出了该进程所依赖的模块。这部分信息仅用于在代码热升级时告知系统该以何种顺序升级各个模块。一般来说，只需列出子进程的主模块，在这里就是tr_server。

内容还真不少；不过谢天谢地，只要做上一次，下次创建监督者的时候就有模板可依了。今后要是想不起来该怎么写子进程规范，你可以随时查阅上述内容。子进程规范的优点在于仅需短短两行代码便可实现大量功能。

搞定了这些，就完工了！这两个小模块费了我们不少口舌，但现在你已经逾越了最大的障碍。至此，我们希望你能够对全局有一个清晰的把握，并明确认识这些内容之间的关系和作用。

4.3 启动应用

你的应用已经完工了：目录结构建立完毕，元数据就绪，应用启动入口编写完成，根监督者也已经实现。来运行你的第一个OTP应用吧。

首先你得编译src目录下的所有代码，从而在ebin目录下生成对应的.beam文件。如果系统设置无误，erlc应位于搜索路径内，假定当前路径已经设置为应用根目录tcp_rpc，那么你可以使用以下命令来完成这个任务：

```
$ erlc -o ebin  src/*.erl
```

（有关erlc的使用请参见2.3.6节。）你也可以将src目录设为当前目录，然后用Erlang shell编译所有模块，再将.beam文件手工移入ebin目录；不过现在你也该学着用用erlc了（配合某种构建工具一起使用效果更佳，例如Make、Cons、SCons或Rake，不过这已经超出了本书的讨论范畴）。

.beam文件就位后，启动Erlang并将ebin目录纳入代码路径，如下：

```
$ erl -pa ebin
```

（-pa是path add的缩写，用于添加单个目录到代码路径的最前方。）在Windows上，请换用werl，参见2.1.1节。

Erlang shell启动后，只需一个命令便可启动应用：以应用名tcp_rpc为参数调用标准库函数application:start/1，如下：

```
Eshell V5.5.5  (abort with ^G)
1> application:start(tcp_rpc).
ok
```

看来启动顺利。没什么好奇怪的——毕竟你已经按部就班地完成了本书布置的所有任务。为了证明应用确实已经启动且运转正常，不妨用3.3节中介绍的方法借助Telnet测试一下。

你可能会奇怪shell是怎么找到各个模块的，你让它启动tcp_rpc应用，但实际上根本没有叫

这个名字的模块。还记得.app文件吧（代码清单4-1）？正如Erlang会在代码路径中搜索.beam文件来加载模块，`application:start/1`函数也会在代码路径中搜索.app文件。由于ebin目录已经位于代码路径之中，shell便可以顺利找到元数据文件ebin/tcp_rpc.app，该文件包含了所需的一切信息——尤其是该用哪个模块（tr_app）来启用整个应用。

你的第一个OTP应用就此大功告成！并没那么难，对吧？这个主题结束之前，还有最后一件事要做：文档生成。

4.4　生成 EDoc 文档

在3.2.2节中，我们解释了如何用EDoc注释来标注代码从而直接从源码中生成文档。如今你的应用已经定义完毕，文档生成就是小菜一碟了。启动`erl`后，在Erlang shell中执行下述命令：

```
2> edoc:application(tcp_rpc, ".", []).
ok
```

现在，你可以用浏览器打开doc/index.html文件来检验生成的结果。你会发现在doc下还生成了一堆其他的文件，但它们都可以通过index.html访问。

请注意，即便源码中没有任何EDoc标注也没关系。这种情况下生成的文档只包含一些基本信息，可用于展示应用中包含哪些模块，每个模块又分别导出了哪些函数，即便如此，也比什么都没有要强多了。

空表`[]`是传给EDoc的额外选项（目前还不需要），句点（.）则表示应用就位于当前目录下。此处应用的位置信息是必需的，通过`-pa`选项传入的相对路径`ebin`无法给系统提供足够的线索来定位应用。但如果退出Erlang shell，将当前目录提到应用目录的上一层，再像这样启动Erlang

```
$ erl -pa tcp_rpc/ebin
```

这时你便可以采用先前调用的函数的一个简化版本来达到同样的目的：

```
Eshell V5.5.5  (abort with ^G)
1> edoc:application(tcp_rpc).
ok
```

现在系统能够自行推算出与tcp_rpc这个应用名相关联的路径了。（详情参见标准库函数`code:lib_dir/1`。）请注意，即便目录名中含有版本号，这个方法也行得通（参见图4-1）。一般来说，代码路径主要由系统中所有已安装应用的ebin目录的绝对路径组成；应用的三位版本号则主要用于构建脚本中。

现在你已经得到了一个完整、可运作的应用，配有一些基本的文档，在此基础之上你还可以通过添加EDoc注释来进一步完善文档。这一章可以圆满结束了！

4.5　小结

在这一章中，我们学习了OTP应用的基础知识，了解了它们的结构，以及如何把代码包装成五脏俱全的应用。我们希望你能紧随相关材料切实实践所有的示例，至少也应该按第3章起始处

的说明下载相关源码，并编译执行。

遵循这些结构并以工业级的OTP库为基础来构建代码，可以令系统的基本容错能力提高一个数量级，并且有助于开发一致、可靠且易于理解的软件。在第10章，我们将讨论另一类软件包，称为发布镜像。发布镜像用于将数个应用聚合成一套完整的Erlang软件服务。到那时，你将了解启动完整Erlang系统的方法；你也将明白为何早先介绍的通过调用`application:start/1`来启动应用的方法只适用于手工测试，而不适用于产品系统。

也就是说，看完前面两章，你便已经向创建产品级质量的Erlang/OTP软件迈出了最重要的一步。在继续讨论本书第二部分中有关如何构建工业级Erlang服务的主题之前，我们先稍作休整。在下一章，我们将向你展示一些用于可视化观察Erlang系统内部情况的标准实用工具。

第 5 章

主要图形化监测工具的使用

本章概要

- 用Appmon和WebAppmon监控应用
- 用Pman管理进程
- 源码级调试器的使用
- 用表查看器检查数据表
- 使用Erlang工具栏

截至目前为止，你已经学习了大量有关Erlang和OTP的知识。我们在上一章介绍了OTP应用和监督者，并探讨了它们在Erlang系统中的工作方式。在本章中，我们将向你展示Erlang提供的若干个用于检视运行时系统的图形化工具，这些工具可以很好地帮助我们增进对系统的理解。借助这些工具，我们可以很好地以图形化方式观察进程、应用和监督层级。我们介绍的第一个工具叫作Appmon，是从应用和监督者的角度来观察系统的专用工具。

5.1 Appmon

顾名思义，Appmon是用来监视OTP应用的工具。它既可以按图形化方式展示系统中当前正在运行的应用及其监督结构；可以查看进程的当前状态；还可以针对这些进程执行一些基本操作。

5.1.1 Appmon GUI

进一步说明之前，我们先启动Appmon。打开Erlang shell并执行`appmon:start()`：

```
Eshell V5.7.4  (abort with ^G)
1> appmon:start().
```

稍后，便会弹出一个类似图5-1的窗口。这就是Appmon的主窗口，其中显示了系统中正在运行着的所有应用，现在这里仅有kernel应用。窗口左侧的指示条显示了整个系统的当前负载。

窗口顶部有一排菜单，你可以通过File菜单中的菜单项退出Appmon，或关闭窗口。在File菜单中还有一个菜单项Show List Box，通过它可以打开另一个窗口，系统中的所有应用会以更精简

的方式在那里列出。当系统中有许多应用运行时，主窗口中会挤作一团，用这个窗口可以更便捷地选择应用。

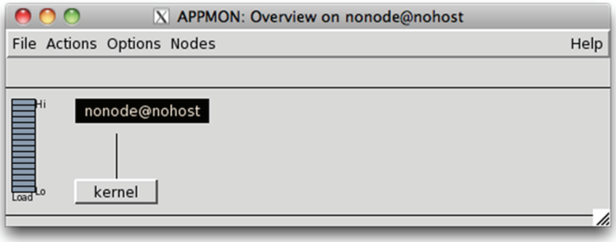


图5-1 Appmon的主窗口。左侧是系统负载指示条（当前为零）。Erlang系统默认只会启动kernel应用

Actions菜单中的菜单项可用于重启Erlang系统（停止并清理所有应用，然后再重新启动这些应用），重新引导整个Erlang系统（重启Erlang虚拟机——为此你必须先配置心跳命令），或停止Erlang系统。这里还有一个名为Ping的菜单项，用于在分布式环境下重建断开的连接。（我们将在第8章讨论分布式Erlang。）

你可以通过Options菜单选择系统负载的计算方式：按CPU时间计算负载，或按CPU等待队列中的进程个数计算负载。在分布式Erlang系统中，你还可以选择节点的显示方式：只用一个窗口每次只显示一个节点，或是同时用多个窗口分别显示所有节点。Nodes菜单中列出的是已知的所有可用节点：当前仅有`nonode@nohost`，即正在以非分布式方式运行着的本地系统。等到学完第8章，你就会明白这些问题。现在你只需要记住，你可以便捷地观察和控制在网络中不同机器上运行着的应用。我们这里不妨启动一个大家熟悉的应用，例如上一章创建的tcp_rpc应用。我们这里按照4.3节的步骤启动它。应用启动完毕后，Appmon主窗口中的kernel应用旁边便会显示出tcp_rpc，如图5-2所示。

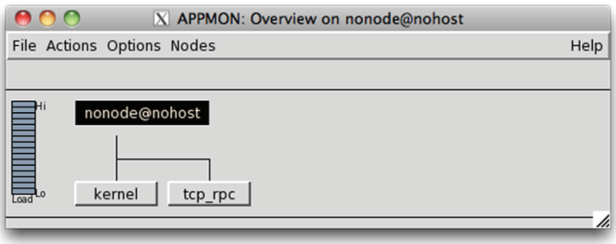


图5-2 启动tcp_rpc应用后的Appmon主窗口。点击应用名称可以看到应用的更多细节

每个应用名都是一个按钮，点击后会打开一个显示应用信息的独立窗口。现在我们单击tcp_rpc按钮。新窗口中显示的正是tcp_rpc应用的监督结构，如图5-3所示。你可能得拉伸一下窗口才能看全所有内容。

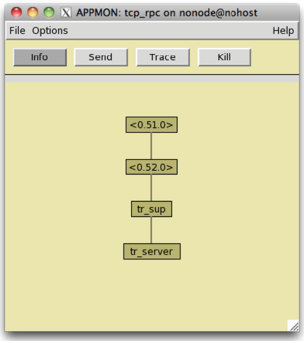


图5-3　Appmon应用窗口，其中展现的是tcp_rpc应用的监督结构。窗口中的按钮用于选定想要执行的动作：Info、Send、Trace或Kill

最上方的两个神秘的匿名进程就是*应用主进程*。它们是application行为模式容器的一部分，由系统在应用启动时派生而来。你无须关心它们，只需要知道它们会调用你的应用行为模式的start函数，在这个例子里就是tr_app:start/2（参见4.1.3节）。相应地，当应用关闭时，它们会在其他进程全部终止后调用tr_app:stop/1。

从上往下数，第三个进程就比较有意思了：它就是你用tr_sup:start_link()启动起来的根监督者进程（参见代码清单4-3）。你会看到它有一个名为tr_server的子进程，这正是由tr_sup:init/1指定的子进程规范中的名字。

应用窗口的顶部有一排按钮。它们用于控制点击窗口中各进程后应该触发什么动作。首先点击按钮选定一个动作模式（默认模式为Info），然后再点击目标进程。选定Info后点击tr_server进程：这样将会打开一个如图5-4的新窗口，其中显示了许多和该进程当前执行状态相关的详细信息。

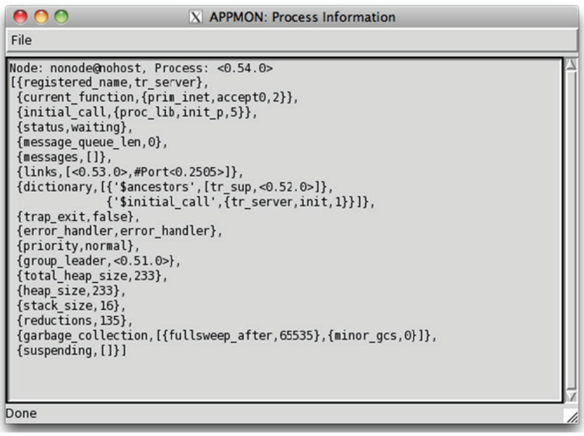


图5-4　Appmon进程信息窗口。它用于展现特定进程的细节信息，诸如消息队列的长度、内存使用量以及当前正在执行的函数

其中消息队列长度、内存使用量以及该进程当前正在执行的函数等信息经常会在调试过程中用到。点击Send按钮后再点击某个进程，会弹出一个小窗口，通过该窗口你可以向选中的进程发送任意Erlang项式（参见图5-5）。你在此处输入的任何项式都会被发送至该进程。

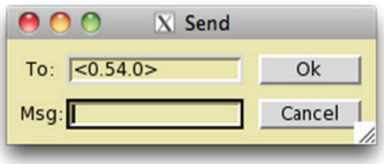


图5-5 Appmon应用窗口中的Send动作可用于向进程发送任意消息。在调试时Send可用于检查进程的反应是否正常；在出现因等待消息而受阻的进程时还可用于解除进程阻塞

下一个动作按钮是Trace。利用它可以跟踪你后续点选的进程。该动作没有什么立竿见影的效果，但它会在后台针对选中的进程启用跟踪。有关进程跟踪设施的详情请参考Erlang/OTP官方文档，既包括由`erlang:trace/3`提供的底层API，以及runtime_tools应用中的`dbg`模块所提供的更为友好的用户接口。关于runtime_tools请参考文档中的工具章节。在5.2节中讨论Pman工具时，我们还会回过头来讨论进程跟踪。

最后一个动作是Kill。该动作会向你点选的进程发送一个不可捕获的`kill`信号。你可以先点选Kill再点选tr_sup进程来测试一下效果。测试的结果应该是tcp_rpc应用的所有进程消失殆尽，只留下一个空白窗口。（回想一下我们曾经讨论过的进程链接、OTP监督者以及1.2节中提到的自动清理机制——由于进程之间的链接方式，终止根监督者会导致整个应用终止。）

5.1.2 WebTool版Appmon

如果乐意（或者你的机器上压根儿就没有图形环境），你可以以另外一种方式来使用Appmon——通过WebTool应用。在Erlang shell下调用`webtool:start()`，你将会看到如下输出：

```
2> webtool:start().
WebTool is available at http://localhost:8888/
Or  http://127.0.0.1:8888/
{ok,<0.62.0>}
```

接着，请打开浏览器并访问http://localhost:8888/（如果是在其他机器上执行，请换用对应的机器名或IP地址）。这将打开WebTool的欢迎页面，在此你可以启动多种工具。其中一个就是WebAppmon，它的界面跟上一节中的GUI类似，只不过需要通过浏览器来呈现。

即便待监视的系统上没有安装Appmon应用（比如仅配备了最小Erlang环境的嵌入式系统），你也可以使用WebTool版Appmon。当前，WebTool版本还不支持停止应用或终止进程。

可以说，Erlang的各种图形化工具各有不同的全局视角。其中Appmon以应用为出发点。我们接下来要考察的工具名为Pman，它的关注点只限于进程，而对应用一无所知。

5.2　Pman

Pman是process manager的简称，这个工具从进程的角度来观察Erlang世界。它可以为你展示系统中当前运行着的所有进程并且允许你对这些进程执行各种操作。

让我们重新启动一个新的Erlang系统并像上一节那样启动tcp_rpc应用。接下来，输入`pman:start()`启动Pman应用：

```
Eshell V5.7.4  (abort with ^G)
1> application:start(tcp_rpc).
ok
2> pman:start().
<0.42.0>
3>
```

Pman启动后，你将会看到一个如图5-6所示的窗口。窗口中列出的是当前运行的进程，同时还显示了进程相关的一些信息，如进程是否经过注册、信箱中的消息数目，以及内存占用量的估算值（以机器字长而非字节为单位）等。其中Reds一列显示的是每个进程的规约（reduction）次数，它大致反映了进程占用的CPU时间。

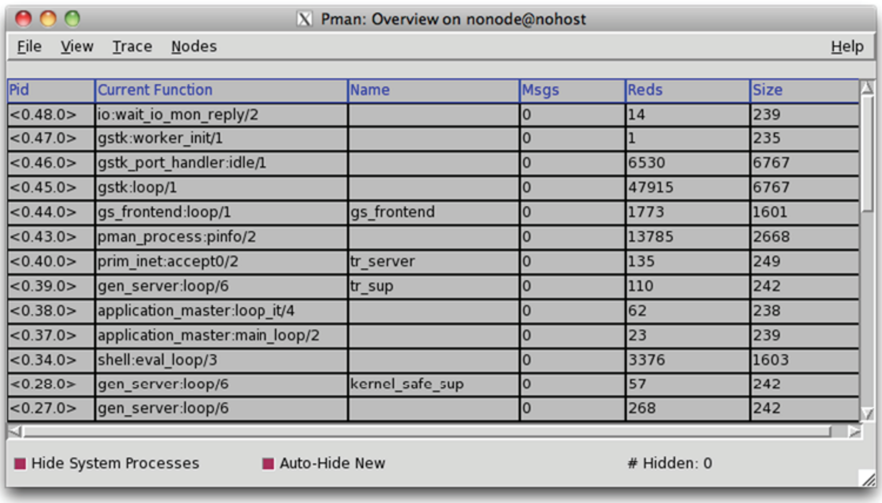


图5-6　显示Erlang系统中所有进程的Pman主窗口，其中还包括注册名、消息队列长度和内存占用量（以字长为单位）等基本信息

你会看到列表中的进程数量颇多。那是因为所有进程全都罗列在了你的眼前，其中也包括正常Erlang环境中的各个系统进程。通常情况下，你只需关心由自己启动的应用中的进程。要精简进程列表，请勾选位于窗口左下角的Hide System Processes复选框。图5-7显示的是隐藏系统进程后留下的你所关心的那些进程。特别是tr_sup和tr_server进程。请跟图5-3中的视图对比一下——在此你是看不出二者之间的联系的。

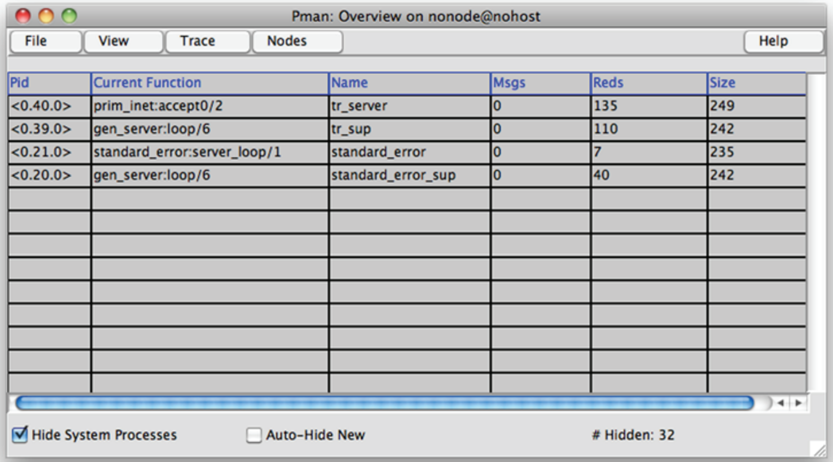


图5-7　隐藏了系统进程的Pman主窗口。这个列表更易于管理。点选View菜单中的Hide Selected Process选项可以进一步精简这个列表

请看第二行（注册名为tr_sup的进程）。最后一次观测该进程时，它正执行到哪儿呢？Current Function一列指示是在`gen_server:loop/6`。可是，这并不像你写的tr_sup监督者的代码呀！怎么回事儿呢？是这样的，程序的这个位置位于我们在3.1.2节介绍过的行为模式容器之中。4.2节中介绍的tr_sup模块是`supervisor`行为模式的一个实现，而`supervisor`行为模式本身又基于`gen_server`行为模式。通常情况下，基于`gen_server`的进程在无事可做时会阻塞在gen_server模块的主循环上，等待着下一条消息的来临。在调试时一定要谨记这一点：进程当前正在执行的函数往往位于通用框架代码之中，因此在很多情况下，进程的注册名更有助于辨别进程。

通过File菜单，你可以退出Pman或设置跟踪选项。和Appmon一样，Pman也有一个用于分布式Erlang的Nodes菜单。Trace菜单则用于启动进程跟踪，其中还包含强制终止进程的菜单项。

目前所有菜单中最有用的一项就是View菜单了。通过它你可以更为细致地筛选要查看的进程，还可以用它刷新窗口中的信息。筛选进程的方式有很多，你可以一上来就隐藏所有的进程再逐个儿把要显示的进程挑出来，也可以先一次性把所有进程都显示出来再把不需要的那些隐藏掉。你还可以根据进程当前执行的模块来隐藏进程，或者在新窗口中显示进程所处的模块的信息。

双击列表中的进程或者选择Trace→Trace Selected Process菜单项，会弹出所选进程的跟踪窗口。该窗口会先显示一些基本信息，然后根据当前选项设置开始跟踪该进程。图5-8展示的是经由File→Options打开的默认跟踪选项的设置窗口。在打开的各个跟踪窗口中也可以单独控制这些选项。

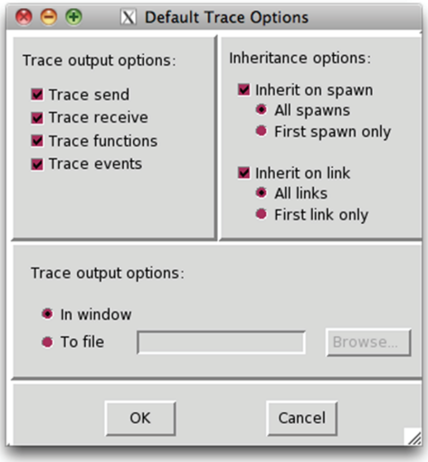


图5-8　Pman的默认跟踪选项设置。你可以选择要跟踪的动作并决定是否跟踪由当前进程新派生出来或链接上的进程。你还可以将跟踪结果转储至文件中

其中，Trace Output区的选项尤为重要。你可以在此决定是否打算查看消息收发、函数调用以及其他各种事件，如进程的派生、链接、终止等。你可以指定如何处理当前跟踪进程新派生出来（或新链接上）的其他进程，此外，你还可以选择将跟踪结果转储到文件而不是GUI窗口。

提醒：如果被跟踪的进程任务繁重，系统执行跟踪的消耗会很高。尤其不要跟踪用于显示跟踪结果的那个系统进程。否则，正如一个企图记录自身行为的日志系统，它很快就会被自己产生的海量信息淹没，并最终导致系统因内存耗尽而崩溃。千万不要在产品环境上胡乱双击列表中的系统进程。（在这儿就无所谓了。）

对于这个例子，请一边尝试用3.3节的方法用Telnet和服务器交互，一边针对tr_server和tr_sup进行一些简单的跟踪。然后再在跟踪tr_sup进程的同时用主窗口Trace菜单中的Kill动作强制终止tr_server进程，观察子进程消亡时监督者的反应。和跟踪相关的内容还有很多，不过仅就了解其工作方式而言，这些已经足够了。

截至目前为止，你已经学会了用Appmon来观察应用以及用Pman来观察进程。下面，我们将用Debugger来观察模块。

5.3　调试器

图形化的源码级调试器仍然是至关重要的开发工具之一，不过跟其他语言相比，开发Erlang程序时很少会用到调试器。原因一方面在于你掌握着更多信息来源，比如日志和崩溃报告。Erlang源码往往又非常清晰（且没有副作用），只要有崩溃报告在手，通常立即就能定位错误。另一方面，如果崩溃报告还不足以定位错误，那么问题多半与复杂的多进程通信、时序、网络和系统负载等因素脱不开干系。在这些情况下，你需要的是良好的日志，这类问题要么无法在图形调试器

里重现，要么由于时序的变化而导致线索尽失。

不过有的时候仍然有对代码进行单步跟踪的必要——比如开发复杂算法或协议时。在Erlang shell中调用`debugger:start()`即可启动调试器：

```
Eshell V5.7.4  (abort with ^G)
1> debugger:start().
{ok,<0.46.0>}
2>
```

该操作将打开主调试器窗口，如图5-9所示。不过这恐怕跟你所预期的调试器不大一样。

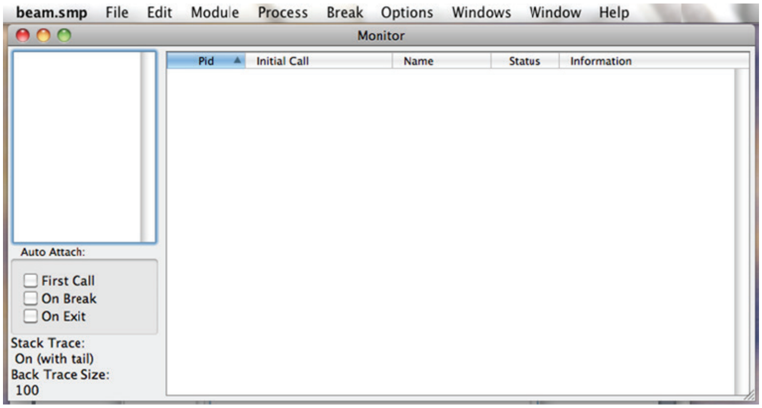


图5-9 源码级调试器启动后的主监视窗口。右侧的大片区域用于显示运行在调试器下的进程，左侧较小的区域用于显示正被解释执行的模块

它跟你所见过的DDD、Eclipse及其他上百种调试器都不一样。原因有二，首先，Erlang采用的架构很可能迥异于你之前所熟悉的架构，它是面向进程的——即便是调试器也是如此：窗口中的大片空白区域用于显示当前连接上调试器的进程。其次，在你明确选择一个模块前它在调试器中是不可见的：左侧的一小块区域用于显示当前选中的模块。

要在调试器中启用某个模块，你必须先让调试器解释该模块。这里涉及两部分内容：调试器既需要.erl源文件（这样它才能显示源码），也需要对应的包含调试信息的.beam文件。你需要在编译时用`debug_info`标志明确告知编译器在.beam文件中加上调试信息。让我们重复4.3节中的步骤重新编译tcp_rpc的代码，这次记得加上这个标志。（这个标志应该直接传给编译器，将它传给`erlc`时应该以+而非-为前缀。）

```
$ erlc +debug_info -o ebin  src/*.erl
```

点选Module→Interpret打开文件选择对话框并点选一个源码文件。调试器会自动从源文件所在的目录中或邻近的ebin目录中找到对应的.beam文件。在这个例子中，请找到tr_server.erl文件并选中它。这样便可以在主调试器窗口的模块列表中看到tr_server。双击列表中的名字可以打开一个用于查看模块源码的新窗口，如图5-10所示。

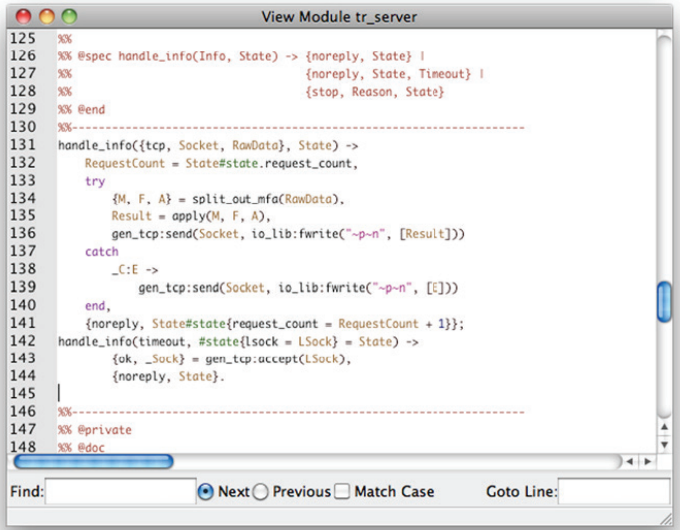


图5-10　显示某模块源码的调试器窗口。你可以在该窗口中对代码进行搜索或跳转到指定行。双击某行可以在改行上设置或移除断点

首先，让我们来设置一个断点。找到代码中的`do_rpc/2`函数，并双击调用`split_out_mfa(RawData)`的这一行，在行号之后，代码行之前会出现一个红色的圈。现在，像之前一样调用`application:start(tcp_rpc)`，并用3.3节中介绍的方法通过Telnet执行一次远程调用。不出所料，没有应答。回过头来看一下主调试器窗口，你会看到名为tr_server的进程被挂在断点处，如图5-11所示。

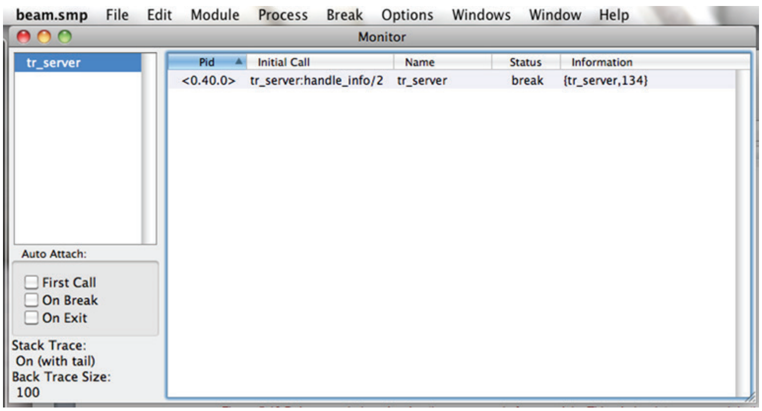


图5-11　主调试器窗口，展现了tr_server中被命中的断点。双击列表中的进程将打开一个可用于跟该进程交互的新窗口

双击主窗口中的tr_server进程，会打开一个形如图5-12的新窗口。通过该窗口你可以与连接到调试器的进程进行交互。如你所见，窗口上有用于单步跟踪、继续执行等功能的按钮，与普通的源码级调试器一样。当前变量的值显示于窗口右下角，单击列表中的变量，完整的值将显示于左下角。在源码中的某行上双击可以在该行上添加或移除断点。请在代码中执行几次单步跟踪，看看会发生些什么。

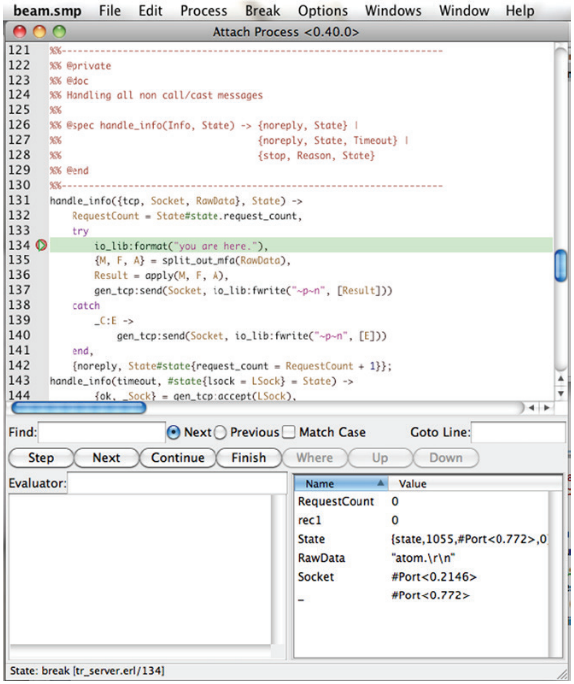


图5-12 调试器窗口，其中显示的是连接到调试器上的某个进程当前的代码位置。你可以在该窗口中单步跟踪代码、增删断点、观察变量等

通过菜单你可以设置条件断点、观察消息队列，或完成一些其他的操作。限于篇幅，我们无法在此详述这些选项，但我们建议你多试试调试器，好好了解一下它的用途。我们希望这段关于调试器的简短介绍能够把你领进门，以便真正将调试器运用到实战中。

现在，再来看点儿别的：一个用于观察数据而非代码的工具。

5.4 表查看器 TV

TV应用跟我们之前讨论的其他工具都不太一样，无论是Appmon、Pman还是调试器，观察的统统都是运行在系统中的代码，TV则用于查看数据。TV是表查看器（Table Viewer）的缩写。它可用于查看Erlang中两种主要类型的表：ETS表和Mnesia表。我们在2.14节介绍过ETS表，在下一章你就会用到它们了。Mnesia数据库则将在第9章介绍。

现在，我们先来简要看看如何查看ETS表。TV启动后的默认视图就是ETS。在Erlang shell中输入`tv:start()`便可启动TV主窗口，如图5-13所示。

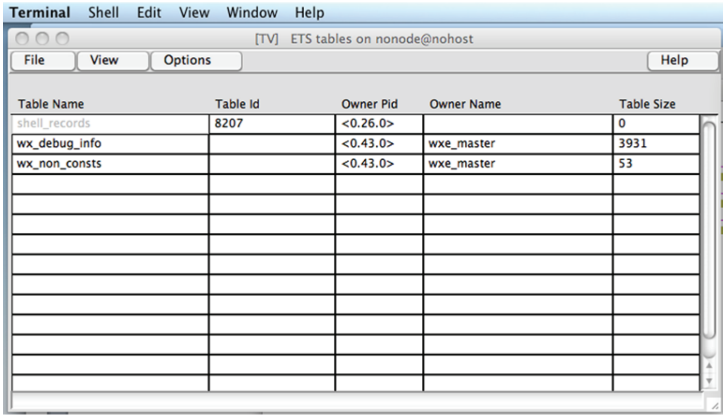


图5-13　TV表查看器的主窗口。默认不显示系统表

在没有运行任何用户应用的Erlang系统中，一开始只有很少几张表，甚至一张表也看不到。点选Options→System Tables菜单项，然后便会显示出一张长长的数据表列表。在列表的顶端，你将找到一个名为`ac_tab`的条目，隶属于application_controller进程。双击该条目（或先单击该条目再点选File→Open Table菜单项）将打开一个形如图5-14的新窗口，其中显示的是表中的内容。

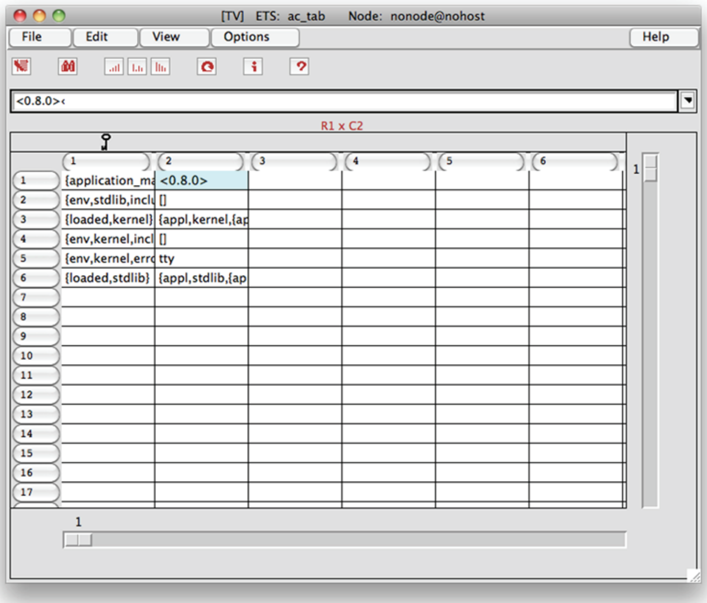


图5-14　TV表窗口。本例中显示的是系统表ac_tab的内容

如你所见，TV以类似电子表格的形式来展示数据表。其中一列带有钥匙标记（位于第一列上方），表示这一列是主键。你可以调整列宽来更好地查看数据内容。这个例子中展示的是隶属于Erlang系统顶层应用控制器的一张系统表。其中包含Erlang的两个核心应用——kernel和stdlib的部分信息（根据系统的不同，可能还含有更多应用中的信息）。

TV应用的界面基本上一目了然，因此我们希望你自行尝试和学习。菜单和图标提供了排序和轮询的选项，你可以获取有关某张表的更为详尽的信息、搜索表项，还可以实时编辑或删除表项。在后续章节中用到ETS表和Mnesia时可不要忘了TV，它可以帮助你更好地观察数据的变化。

4个GUI工具的介绍就到此结束了。虽然它们功能各异，却都有一个共同点：都可以从Erlang工具栏启动。

5.5　工具栏

工具栏应用是一个小窗口，窗口中的每个图标对应于一个应用。如果要频繁使用这些工具，例如在调试时，比起像之前那样独立启动各个应用，启动工具栏并把它丢到桌面的一角会更为便捷。在Erlang shell中调用`toolbar:start()`，便会看到如图5-15的窗口。

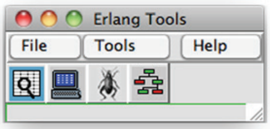


图5-15　Erlang工具栏。如果需要频繁启动TV、Pman、Debugger或Appmon，工具栏会很方便。你甚至还可以向工具栏上添加自定义图标

第一个按钮可以启动TV，下一个可以启动Pman，第三个可以启动Debugger，第四个可以启动Appmon。另外，虽然并不常用，但你也可以向工具栏上添加用于启动其他自定义应用的按钮：先点选Tools→Create Tool File，然后填写新工具的详细信息，包括你要使用的图标文件以及用于在点击图标后启动工具的模块和函数。例如，指定mymod和myfun将调用`mymod:myfun()`。（工具启动函数不接受任何参数。）

5.6　小结

我们在此介绍了用于观察运行中的Erlang系统并与之进行交互的主要可视化工具。我们希望这些内容足以帮助你把它们变成你的日常工具。现在你可以继续自行深入探索了。在官方Erlang/OTP文档的工具一节可以找到更多相关内容。

Part 2

第二部分

构建生产系统

欢迎回到真切的大千世界，这里才是 OTP 的用武之地。在本书的这一部分中，我们将跟随 Erlware 团队运用 Erlang，特别是 OTP，来解决他们所面临的一系列问题。这一部分的内容将涵盖行为模式、监控、打包以及许多其他的用于构建工业级 Erlang 软件的关键手法和技术。

第 6 章

打造一套缓存系统

本章概要

- ❑ 设计一个简单的缓存服务
- ❑ 建立基本的应用结构与监督结构
- ❑ 实现缓存的主要功能

现在你已经对Erlang/OTP有了基本的了解，该向更高级也更现实的案例迈进了。从本章开始，本书的第二部分将带你一起搭建一个实用的分布式Erlang应用。为了明确这些任务背后的动机，我们将以一个名为Erlware的开源项目为背景来讲述整个开发过程，这个项目正面临着一系列假想但却富有现实意义的挑战，你的任务就是解决这些难题。

6.1 故事背景

Erlware项目旨在让人们更便捷地获取所需的Erlang应用，从而为应用的终端用户以及那些期望将应用用到自己的项目中的开发者提供便利。随着Erlang的日益流行，便捷地获取开源应用及构建工具的需求也越来越旺盛。这给Erlware项目带来了迅猛增长，但与此同时用户的抱怨声也响了起来，项目管理员们发现网站当前的响应速度已经无法满足用户的需求了。各种怨言中针对网页加载速度的最多。不过，Erlware团队十分关注用户体验，更何况有传言说Google等搜索引擎会降低页面加载速度慢的站点的权重。

用户抱怨最多的就是软件包搜索页面。在该页面上，用户可以从庞大的Erlware库里搜索特定的Erlang/OTP软件。为了生成这个页面，Web服务器必须访问一组软件包服务器并向每台服务器请求一份软件包列表，每台服务器所保存的软件包列表的内容互不相交。软件包服务器之间相互独立，也不存在持有所有软件包数据的中央数据库。早先Erlware的规模还小，仅有一台软件包服务器，这个架构没什么问题。即便后来服务器增加到了3台，情况也都还好，但再加服务器的话就不行了。该系统的结构如图6-1所示。

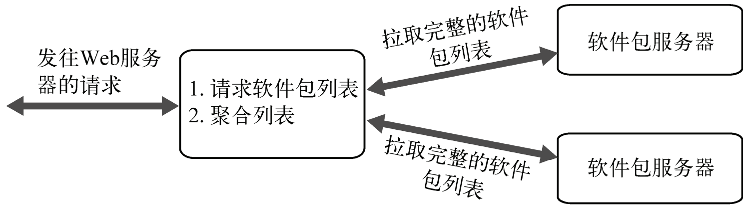


图6-1　Erlware当前的架构：Web服务器在处理每个请求时都会从所有软件包服务器上抓取一份完整的列表

Erlware团队的成员聚在一起讨论加速方案，最终决定通过给Web服务器添加本地缓存来提速。这样一来，查询软件包列表的同时，软件包服务器返回的列表将以URL为键存入缓存；随后，当用户访问同一个URL时，直接从缓存中取出软件包列表便可迅速完成页面渲染。该架构如图6-2所示。

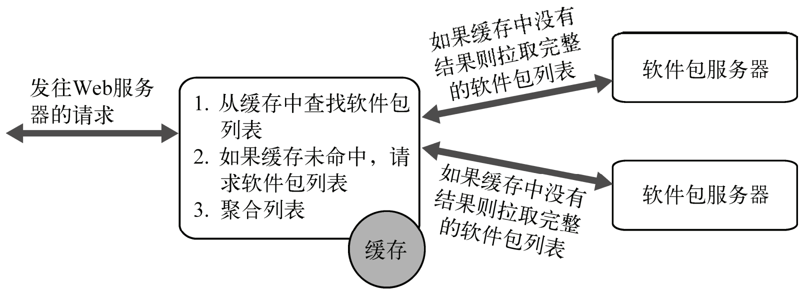


图6-2　规划中的新架构。Web服务器将软件包列表放入本地缓存，这样就无须在处理每一个请求时都做一遍查询了

该缓存服务将以独立OTP应用的形式来实现。本章后续内容便以该缓存系统的基本功能为中心，这些功能大体包括：

- 缓存的启动和停止；
- 向缓存中添加键/值对；
- 查询与给定的键相对应的值；
- 更新与给定的键相对应的值；
- 删除键/值对。

有了这些，你便能够搭建出一个极为简单但五脏俱全的缓存服务。本章中搭建的是该缓存服务的第一个版本，它没有什么高级功能，只是一个可以通过普通Erlang函数调用来访问的独立缓存服务，用于加速单台服务器上的单个Web服务器。分布式支持等高级特性将留待后续章节实现。在着手实现这个基本的缓存服务之前，我们不妨先来讨论一下设计方案。

6.2　缓存的设计

这个简易缓存存储的是键/值对，其中键与键之间不得重复，并且每个键只能映射到一个值。

这个设计背后的核心思想是为写入缓存的每一个值都分配一个独立的存储进程，再将对应的键映射至该进程。你可能会对这种为每个值分配一个进程的设计感到惊讶，甚至觉得不可思议；但对缓存这类服务而言，这个设计是合理的，因为缓存中的值相互独立，各有各的生命周期。同时，Erlang本身对大量轻量级进程提供了良好的支持，使得这种设计成为可能。

为了搭建这个缓存，你得先建立一些基本的子系统，其中每个子系统都是一个独立的模块。图6-3展示了该简易缓存的各个组成部分。

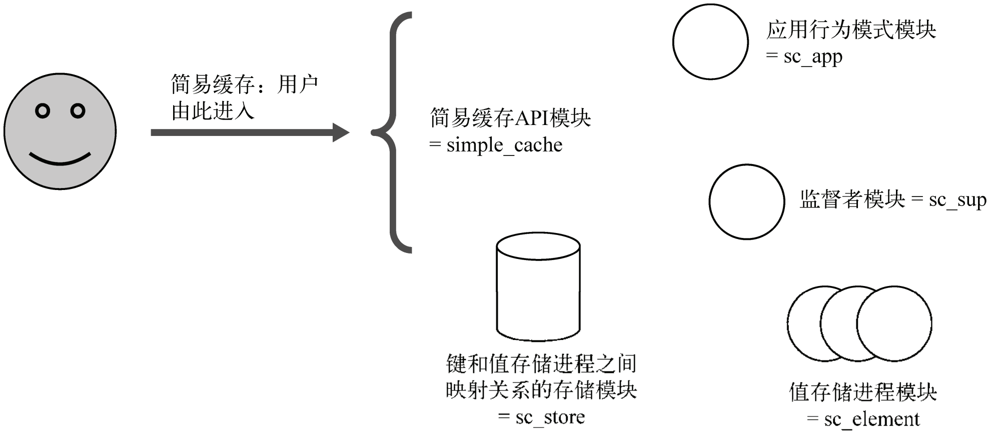


图6-3　简易缓存的各个部件。包括API前端、主应用模块、若干数据存储进程、监督者模块，以及负责维护键和数据存储进程之间映射关系的映射表

如图6-3所示，你一共需要创建5个模块，详情参见表6-1。

表6-1　简易缓存应用中的模块

模　　块	用　　途
simple_cache	用户API；应用的外部接口
sc_app	应用行为模式实现模块
sc_sup	根监督者实现模块
sc_store	用于封装键和pid之间映射关系的模块
sc_element	缓存数据存储进程

模块命名规范

还记得我们在 3.2 节提到过的模块命名规范吗？这个应用中除主用户 API 模块 simple_cache 以外，所有模块都以 sc_（代表 Simple Cache）为前缀。OTP 应用往往有一个与应用同名的前端模块，这种模式很常见。

用户只能通过simple_cache API模块与缓存服务交互。该模块直接与sc_store模块和sc_element模块通信，前者负责维护键和进程间的映射关系，后者负责存储进程的创建、更新和删除。所有

存储进程都受sc_sup监督进程监督，除此之外，还有一个负责整个缓存系统的启动和停止的应用行为模式模块sc_app。图6-4从进程和数据流的角度展示了该架构。

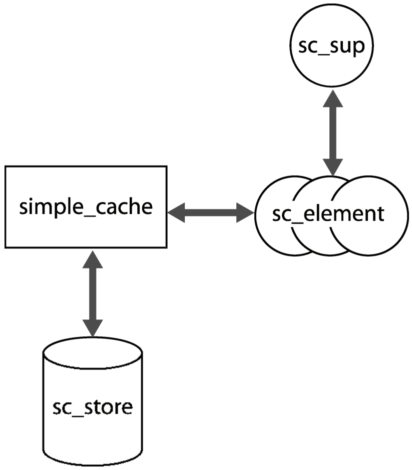


图6-4　模块及相应进程间的数据流。用户API模块simple_cache仅跟sc_store和sc_element模块直接通信

在处理运行时的键/值对插入操作时，监督进程会按需派生sc_element进程。派生出来的进程会记住与给定的键相关联的值，随后，sc_store会记录下该键与该进程ID间的映射关系。键/值之间的映射关系就这样建立起来了。要获取与指定的键相关联的值，首先应该查找与键相关联的存储进程的ID，然后再向该进程查询当前持有的值便可。图6-5是这层间接映射关系的图解。

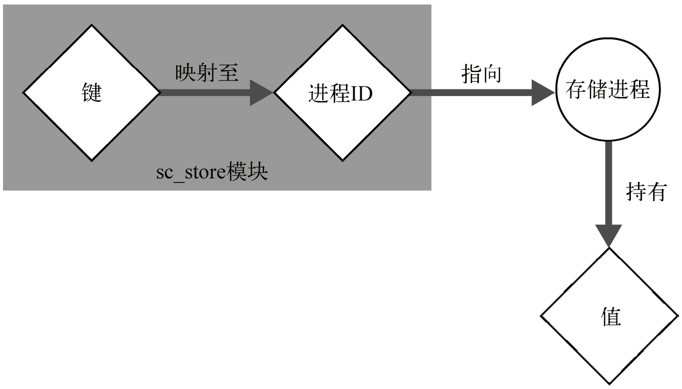


图6-5　键/值之间的间接映射。每个值都由单独的进程负责存储

这些想法目前看来可能有些另类，但这个架构可以有效发挥出OTP中很多强大机制的威力。待到本章结束之时，你将会发现自己无偿获得了多么丰富的功能，而且所有这一切仅需寥寥数行代码而已。闲话少说，让我们先来搭建缓存服务的应用基础架构吧。

6.3　创建 OTP 应用的基本骨架

规范的OTP应用是一切优秀Erlang项目的立足点。一个项目通常可以分解成多个应用，不过，仅就目前的简易缓存服务而言，一个应用就足够了（当然，在设计上它仍然可以被用其他项目的组件）。

这一节将沿用第4章中搭建tr_server应用的方法。但这次我们换个思路：先不考虑任何实际功能，仅从设计角度出发搭出一个应用骨架，之后再逐步添砖加瓦。

概括来说，应用结构的搭建分为以下几个步骤：

(1) 创建标准应用目录布局；

(2) 编写.app文件；

(3) 编写应用行为模式实现模块，即sc_app；

(4) 实现顶层监督者，即sc_sup。

我们所要搭建的是个具备自主运转能力的主动应用，为此我们还需要准备一个应用行为模式的实现模块和一个根监督者。遵循第4章中的规范，这两个模块的名称应该分别以_app和_sup结尾。不过在此之前，得先建好目录结构。

6.3.1　应用目录结构的布局

首先新建一个名为simple_cache的顶层应用目录。在该目录下，如4.1.1节介绍的那样，新建doc、ebin、include、priv和src等子目录。最终的目录树应该是这样的：

```
simple_cache
            |
            | - doc
            | - ebin
            | - include
            | - priv
            | - src
```

（这个例子还用不上doc、include和priv目录，不过，反正多建几个目录没有什么坏处，以备不时之需吧。）目录布局完毕后，下一步就是布置.app文件。

6.3.2　创建应用元数据

如4.1.2节所述，在启动应用或在执行运行时代码热升级时，OTP需要了解一些用于描述应用自身的元数据。存放元数据的.app文件的文件名应该与应用名相匹配（但无须采用特定模块的名字），在这个例子中，该文件就是ebin/simple_cache.app。.app文件当前内容如下：

```
{application, simple_cache,
 [{description, "A simple caching system"},
  {vsn, "0.1.0"},
  {modules, [
             sc_app,
             sc_sup
            ]},
```

```
    {registered, [sc_sup]},
    {applications, [kernel, stdlib]},
    {mod, {sc_app, []}}
   ]}.
```

与代码清单4-1作个比较，你会发现二者很相似。sc_app和sc_sup这两个模块肯定是少不了的，已经罗列在内；其余模块则将在后续逐步加入该列表。另外，很明显根监督者应该以sc_sup为注册名进行注册。

骨架已经搭建完毕，接下来就要给应用行为模式的实现模块添砖加瓦了。

6.3.3 实现应用行为模式

应用行为模式的实现位于文件src/sc_app.erl内，如代码清单6-1所示。不妨将之与4.1.3节中的tr_app.erl作个比较。特别需要注意的是，.app文件中的mod元组给出了应用行为模式模块的模块名，系统就是从这里得知应该从何处启动和停止应用的。

代码清单6-1 src/sc_app.erl

```
-module(sc_app).

-behaviour(application).            <-- 行为模式声明

-export([start/2, stop/1]).         <-- 导出的行为模式回调函数

start(_StartType, _StartArgs) ->
    case sc_sup:start_link() of     <-- 启动根监督者
        {ok, Pid} ->
            {ok, Pid};
        Other ->
            {error, Other}
    end.

stop(_State) ->
    ok.
```

sc_app模块唯一的任务就是在应用启动时启动根监督者（在6.4.2节中会对这个地方稍作修改）。在应用停止时则什么也不用做。

6.3.4 实现监督者

根监督者在文件src/sc_sup.erl中实现（参见代码清单6-2）。此处用到的监督者与第4章中创建的监督者不同；我们没有给这个监督者静态指派任何永久子进程，但却可以给它动态添加任意多个同类型的临时子进程。

代码清单6-2 src/sc_sup.erl

```
-module(sc_sup).

-behaviour(supervisor).

-export([start_link/0,
         start_child/2              <-- 动态启动子进程
```

```
        ]).

-export([init/1]).

-define(SERVER, ?MODULE).

start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

start_child(Value, LeaseTime) ->
     supervisor:start_child(?SERVER, [Value, LeaseTime]).

init([]) ->
    Element = {sc_element, {sc_element, start_link, []},
               temporary, brutal_kill, worker, [sc_element]},
    Children = [Element],
    RestartStrategy = {simple_one_for_one, 0, 1},
    {ok, {RestartStrategy, Children}}.
```

sc_element:start_link/2的参数

1. 简易一对一监督

该监督者的监督策略被设定为simple_one_for_one（简易一对一监督）。采用诸如4.2.1节中用过的one_for_one等其他重启策略时，监督者一般需要同时管理多个与自己同时启动的子进程，通常这些子进程的生命周期也与监督者相同。simple_one_for_one型监督者只能启动一种子进程，但却可以启动任意多个。它所有的子进程都是运行时动态添加的，监督者本身在启动时不会启动任何子进程。

观察代码清单6-2中的监督者模块，它的init/1函数与代码清单4-3中的init/1很相似。此前虽然我们也只用到了一个子进程规范，但现在却只能有一个：simple_one_for_one型监督者的init/1必须指定一种且仅一种子进程，但子进程并不会随监督者一同启动。不过，你随时可以通过调用简化版supervisor:start_child/2函数，令监督者启动新的子进程。其他类型的监督者在动态添加子进程时，必须将完整的子进程规范传递给start_child/2。但对于simple_one_for_one型监督者而言，由于所有子进程都遵循同一套已知的子进程规范，你只需要说一声“再来一份”就可以了。这套机制恰恰可以满足你当前的需求。

2. 监督者模块

sc_sup模块有两个API函数，代码清单4-3中的tr_sup则只有一个。前文已经描述了新出现的start_child/2函数的作用：令运行中的监督者（由?SERVER标识）启动一个新的子进程，并将Value和LeaseTime参数传给子进程的入口函数（因为每个子进程的这两个参数各不相同）。将这些逻辑组织成一个API函数将更有利于模块中实现细节的封装。

init/1函数与代码清单4-3中的版本有些细微的差别，请仔细阅读并参阅4.2.3节比较一下个中细节。二者最核心的区别显然在于重启策略的不同，此处采用的策略是simple_one_for_one，重启执行频率最多为每秒0次（即不执行重启）。这个监督者的子进程被标为temporary型而非permanent型，也就是说子进程退出后无须重启。在很大程度上，这个监督者不过是sc_element进程的一个工厂而已。此外，此处的关闭策略也被设置成了brutal_kill，表示子进程应随监督者的关闭而立即终止。（对于simple_one_for_one型监督者而言，监督者无须费心

主动关闭子进程；一收到监督者终止时触发的退出信号子进程便会立即自行终止。只要子进程遵循标准的OTP行为模式，这一点便确凿无疑。此处之所以还要写明brutal_kill，主要还是用于明确表明意图。）

调用start_child/2 API函数时，当前进程会向监督进程发送一条消息，令它以Value和LeaseTime为参数调用sc_element模块的start_link函数，进而启动一个新的子进程。子进程规范中的元组

```
{sc_element, start_link, []}
```

给定了模块名、函数名和子进程启动函数的参数，调用start_link之前，列表[Value, LeaseTime]将被并入参数列表[]，从而形成最终的函数调用sc_element:start_link(Value, LeaseTime)。

每调用一次sc_sup:start_child/2，就会新启动一个带有自己的值和淘汰时间的sc_element进程。这就形成了一棵动态生成的监督树，如图6-6所示。

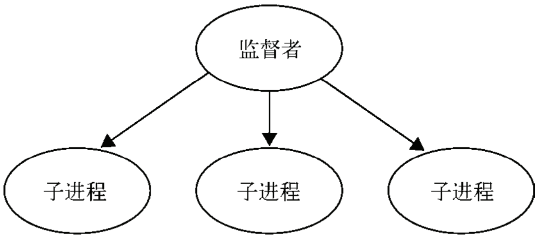


图6-6　简易一对一监督的层级结构。所有子进程同为一个类型，可动态增删且数量任意

至此，一个可运行的应用骨架就搭建完毕了。你可以从Erlang shell中启动它并观察它的运作情况。当然，目前除了启动和停止应用以外你还什么都做不了，因为应用的实际功能和用户接口都还没有实现。由于采用了simple_one_for_one型监督策略，监督者在启动时不会启动任何子进程；此外，由于sc_element尚未实现，调用sc_sup:start_child/2会触发运行时错误。本章后续内容将逐步添加这些功能，最终实现一个羽翼丰满的simple_cache应用。

6.4　从应用骨架到五脏俱全的缓存

在继续学习之前，请再看一看图6-3所示的Simple Cache应用的设计思路，同时请回顾一下表6-1中列出的待实现的模块。没错，应用中最基本的sc_app和sc_sup已经完成了。剩下的还有：

- simple_cache——用户API；
- sc_element——用于存储缓存数据的进程；
- sc_store——用于维护键与进程间的映射关系。

接下来，首先要实现sc_element模块，这样顶层的监督者才有子进程可供启动。随后要实现的是sc_store，用于建立键与进程间的映射关系，最后我们还需要一个API模块，从而为整个应用提供一套良好的接口。

6.4.1 编写sc_element进程

sc_element模块实现了sc_sup的子进程，每当有新数据插入缓存时，sc_sup就会派生出一个新的sc_element进程，用于存储与给定的键相关联的数据。我们打算以gen_server行为模式为基础来实现这类进程（类似于第3章中的tr_server），数据将被保存在gen_server进程的进程状态中。gen_server的工作原理请参阅第3章，此处不再赘述。

1. 模块首部

代码清单6-3所示的模块首部与代码清单3-2类似，你应该已经很熟悉了。二者的主要区别在于API，这是理所当然的——虽然内部都基于同一框架，两个服务器的用途却不尽相同。你要实现的主要功能有四个：新元素创建、元素值查询、元素值替换，以及元素删除。

代码清单6-3 src/sc_element.erl的首部

```
-module(sc_element).

-behaviour(gen_server).

-export([
         start_link/2,
         create/2,
         create/1,
         fetch/1,                          导出的API函数
         replace/2,
         delete/1
        ]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).                           一天中的总秒数
-define(DEFAULT_LEASE_TIME, (60 * 60 * 24)).
                                                          状态记录
-record(state, {value, lease_time, start_time}).
```

键/值对在缓存中存活一段时间之后便会被清理出局，这段时间称作淘汰时间。DEFAULT_LEASE_TIME便是默认的淘汰时间（以秒为单位）。设定淘汰时间的目的在于保证缓存中的内容足够新，缓存就是缓存，不是数据库。创建sc_element进程时，你可以通过API自行调整这个值。

模块首部的最后一项定义了用于表示gen_server进程状态的记录。它由3个字段组成：进程持有的值、淘汰时间，以及进程启动时的时间戳。

2. API段和进程的启动

模块中的下一部分就是API的实现，如代码清单6-4所示。

代码清单6-4 src/sc_element.erl的API段

```
start_link(Value, LeaseTime) ->
    gen_server:start_link(?MODULE, [Value, LeaseTime], []).

create(Value, LeaseTime) ->
```

```
    sc_sup:start_child(Value, LeaseTime).                    <— 将启动委托给sc_sup

create(Value) ->
    create(Value, ?DEFAULT_LEASE_TIME).

fetch(Pid) ->
    gen_server:call(Pid, fetch).

replace(Pid, Value) ->
    gen_server:cast(Pid, {replace, Value}).

delete(Pid) ->
    gen_server:cast(Pid, delete).
```

正如我们在6.3.4节说明的那样，子进程由监督者负责创建；个中细节则由监督者的API函数sc_sup:start_child/2负责屏蔽。然而，监督者的存在属于实现细节，sc_element的用户并不关心。为此，我们创建了API函数create/2，用于将创建子进程的任务委托给sc_sup。此外，如果直接采用默认淘汰时间，还可以选用更为简化的create/1。这下即便把底层实现改个底儿朝天，也不用动接口层了。

回顾一下，我们在6.3.4节为simple_one_for_one型监督者设定了子进程规范，于是每当sc_sup:start_child/2创建新的子进程时，它都会回调sc_element:start_link(Value, LeaseTime)。与代码清单3-3中的tr_server:start_link/1类似，这个API函数将会进一步调用更为标准的函数（这里指gen_server:start_link/3）。但是在这里，最终用户不应该直接调用start_link/2（那样的话启动起来的进程将无法接受监督），也无须劳烦gen_server库为你完成进程注册（因为sc_element进程数量众多，不是单例）。

这个调用流程很是令人费解，让我们再来复习一遍，彻底搞清楚来龙去脉。插入新元素时，应调用sc_element:create(...)，该调用会将任务委托给监督者API函数sc_sup:start_child/2，进而由监督者调用库函数supervisor:start_child/2。借助子进程规范及附加的调用参数Value和LeaseTime，监督者的代码又会回调sc_element:start_link/2。这里还没有讲到sc_element进程的实现方式，不过由于该模块以gen_server为基础，调用流程至此将被转交给库函数gen_server:start_link/3，并最终将新的子进程启动起来。整个调用流程如图6-7所示。

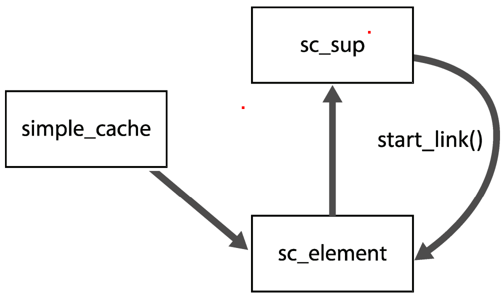


图6-7 存储新存储时的调用流程。其中sc_element API向simple_cache屏蔽了sc_sup的存在。同时，sc_sup也不关心sc_element的功能细节

其余的几个API函数——`fetch/1`、`replace/2`和`delete/1`——都比较简单，和代码清单3-3中tr_server的API函数差不多。它们的任务无非就是利用`call`或`cast`向进程发送请求。其中又只有`fetch/1`需要等待应答，因此其他两个API可以采用异步的`cast`从而立即返回至调用方。这个模块与tr_server有一个细微而关键的差别，就是所有sc_element进程都没有注册名。（因为它们的数量不受限制。）也就是说API函数必须包含进程标识符，否则`gen_server`的函数就没法知道该将消息发往何处了。当然，这也就意味着客户端必须自行维护这些标识符。等我们在6.4.2节中实现sc_store模块时再回过头来看这个问题。

3. gen_server回调段

sc_element进程启动之后的第一件事就是通过gen_server回调函数`init/1`完成进程初始化，有关gen_server回调的工作机制请参阅3.2.4节。该函数应返回一个初始化完毕的进程状态记录。`gen_server:start_link/3`调用将一直阻塞到`init/1`返回为止。src/sc_element.erl的各个回调如代码清单6-5所示：

代码清单6-5　src/sc_element.erl中的gen_server回调段

```
init([Value, LeaseTime]) ->
    Now = calendar:local_time(),
    StartTime = calendar:datetime_to_gregorian_seconds(Now),
    {ok,
     #state{value = Value,                          <- 初始化进程状态
            lease_time = LeaseTime,
            start_time = StartTime},
     time_left(StartTime, LeaseTime)}.              <- 初始化超时设置

time_left(_StartTime, infinity) ->
    infinity;
time_left(StartTime, LeaseTime) ->
    Now = calendar:local_time(),
    CurrentTime =  calendar:datetime_to_gregorian_seconds(Now),
    TimeElapsed = CurrentTime - StartTime,
    case LeaseTime - TimeElapsed of
        Time when Time =< 0 -> 0;
        Time                -> Time * 1000
    end.

handle_call(fetch, _From,  State) ->
    #state{value = Value,
           lease_time = LeaseTime,
           start_time = StartTime} = State,
    TimeLeft = time_left(StartTime, LeaseTime),
    {reply, {ok, Value}, State, TimeLeft}.          <- ❶ 取出进程状态中的值

handle_cast({replace, Value}, State) ->
    #state{lease_time = LeaseTime,
           start_time = StartTime} = State,
    TimeLeft = time_left(StartTime, LeaseTime),
    {noreply, State#state{value = Value}, TimeLeft};
handle_cast(delete, State) ->
    {stop, normal, State}.                          <- ❷ 发出关闭信号

handle_info(timeout, State) ->
```

```
    {stop, normal, State}.

terminate(_Reason, _State) ->
    sc_store:delete(self()),          ❸ 删除进程的键
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

设置服务器超时

如果忘了在回调函数的返回值中设置新的超时，超时将被重置为 inifinity。因此一旦用上服务器超时，切记在每个回调函数的每个子句中都设置好超时。

以下是这些回调函数的详细说明。

- init/1——init/1函数并不复杂。该函数调用了一组标准库函数来获取进程的启动时间，进而将之转换成了公历秒数：也就是按西方/国际公历计算，自公元0年（即公元前1年）至今的秒数，这是一种实用而统一的时间表示法。详情请参见标准库中calendar模块的文档。函数的其余部分负责填写淘汰时间和启动时间等服务器状态字段。

  请注意函数所返回的元组，其中第三个元素设置了初始化完成后的服务器超时。你应该还记得，在第3章的tr_server中也采用了同样的手法，但目的却有所不同。此处的超时用于管理淘汰时间。你将会看到，一个进程存储一个值的设计决策可以大大简化淘汰时间的管理，实现起来会简单很多。如果服务器进程在淘汰时间到期之前一直无人问津，就会收到一条timeout消息，该消息随即被传递给handle_info/2函数，进而令进程关闭。

  工具函数time_left/2用于计算当前离淘汰时间超时还剩多少毫秒，注意系统采用的时间单位是毫秒而不是秒。其中淘汰时间参数既可以是整数也可以是原子infinity；另外该函数绝对不会返回负数。
- handle_call/3:fetch——调用fetch/1 API函数的目的是获取一个值，因此应该向服务器进程发起一个同步的call调用。该调用最终会传递至handle_call/3回调函数。

  你只需要用模式匹配提取出进程状态中的值并将之回传给调用方❶就可以了（参见fetch/1返回的元组，其中的{ok, Value}就是）。唯一麻烦的是你还得用启动时间和淘汰时间推算出新的超时值。至于进程状态，原封不动地返回给gen_server即可，无须作任何修改。
- handle_cast/2:replace——API函数replace/2用于更新现有sc_element进程所持有的值。这个操作无须给客户端应答，因而可以使用异步的cast，该调用最终会转至handle_cast/2。与往常一样，消息内附带一个标记，该标记与发送消息的函数同名。在创建具有复杂接口的服务器时，这个简单的惯用法可以为你免除很多麻烦。

  代码清单6-5中handle_cast的第一个子句负责处理{replace, Value}消息。这部分代码与fetch类似，但这里无须提取原有的值，也无须发送应答，只需将用于替换原值的新值存入进程状态并返回给gen_server即可（有关记录更新的更多知识请参见2.11.3节）。

- handle_cast/2:delete——API delete/1同样没有返回值，因此也可以使用cast，并最终归结到handle_cast/2的第二个子句。delete只需终止sc_element进程，从而将之从缓存中移除即可。

  请注意delete和replace返回的两个元组之间的区别。后者返回的是noreply，表示服务器不用发送任何应答继续运行就好，前者返回的却是stop，而这会导致gen_server自行终止❷。进程终止的原因被设定为normal。这个特殊值用于通知OTP系统服务器是正常关闭的：除非进程是permanent型，否则不要进行重启，也不要在日志中记上一笔非正常关闭。
- terminate/2——gen_server关闭时，会调用terminate/2回调来执行一些清理操作。在这个例子中，你得抹除与进程相关联的键❸。进程终止时，进程状态也随之消失，然而进程状态仅负责值的存储，不负责维护键与进程ID间的映射关系。我们曾经提过，这层映射关系由sc_store模块负责，该模块将在下一小节实现。在此你只需要调用sc_store:delete(Pid)即可，参数就是即将关闭的这个sc_element进程的pid。

sc_element及其API函数的实现就介绍完了，总结如下：

- start_link/2；
- create/2（以及create/1）；
- fetch/1；
- replace/2；
- delete/1。

这些函数相互协作，共同实现了sc_element进程的数据存取、检索、更新及删除功能。在删除操作中，你与sc_store模块打了一个照面，它负责维护键与pid间的映射关系，我们将在下一节介绍该模块的实现。

6.4.2　实现sc_store模块

至此，你已经实现了基本的应用结构和缓存的后端存储系统，其中还包括淘汰时间管理功能。现在，你将以此为基础构建出一整套完整的存储系统。目前，所有缺失环节中最关键的一环就是键与进程标识符之间的映射关系（如图6-5所示），有了它你才能根据给定的键查找到相应的值。该映射的实现将用到ETS表（Erlang Term Storage，参见2.14节），随着工程的深入，我们将陆续介绍有关ETS表的各种细节。

系统的其他部分无须关心存储由ETS实现的事实。sc_store模块扮演了一个抽象层的角色，屏蔽了实现键与pid间映射关系的存储机制。除了使用ETS，该功能还可以通过一个将映射数据保存在进程状态中的gen_server进程实现；也可以在每次插入、删除数据时都将数据写入的某个磁盘文件；甚至还可以用关系型数据库。但无论怎样实现，都应该将应用本身从存储系统的选型中解耦，从而给今后的改进方案留下一些回旋余地。有了这重屏蔽，你甚至可以在不修改其余应用代码的情况下迁移至支持数据冗余的数据库存储方案。

快速回顾一下第2章中的相关内容，ETS表是用于存储Erlang数据的高速内存哈希表。ETS用

C写成，是Erlang运行时系统（ERTS）的一部分，可经由一组内置的Erlang函数访问。表中的每个表项都是一个元组，该元组的一个字段被用作键（通常是第一个字段）。ETS表尤其适合符合以下条件的数据存储需求：

- 无须在多个虚拟机间共享；
- 需要持久化，且仅需要在VM运行期间持久化；
- 由VM内的多个不同进程共享；
- 访问速度要快；
- 数据结构相对平坦，最好不要与其他表有外键关系。

缓存系统的存储需求正好与上述条件相符：前文所述的映射关系需要在缓存服务运行期间持久化，VM关闭后则不再必要，这些数据也不需要在多个VM间共享；进程之间需要共享数据表；数据读取速度要快，因为查询操作位于缓存服务的关键路径上；另外数据本身的结构也很扁平，并且与其他表无关。本质上它就是一张由一对对的键和进程标识符组成的数据表。

代码清单6-6展示了src/sc_store.erl的代码。这一次，这个模块没有采用任何OTP行为模式，也没有与任何进程相关联——它只包含一组供其他进程调用的库函数。（不过后续还会有所变化。）

6

代码清单6-6　src/sc_store.erl

```
-module(sc_store).

-export([
         init/0,
         insert/2,
         delete/1,
         lookup/1
        ]).

-define(TABLE_ID, ?MODULE).
init() ->
    ets:new(?TABLE_ID, [public, named_table]),
    ok.

insert(Key, Pid) ->
    ets:insert(?TABLE_ID, {Key, Pid}).

lookup(Key) ->
    case ets:lookup(?TABLE_ID, Key) of
        [{Key, Pid}] -> {ok, Pid};
        []           -> {error, not_found}
    end.

delete(Pid) ->
    ets:match_delete(?TABLE_ID, {'_', Pid}).
```

API由`init/1`（负责存储系统的初始化）和处理基本CRUD操作（创建、读取、更新和删除）的3个函数组成，其中`insert/2`同时负责创建新表项和更新现存表项。这几个函数的实现都很简洁。我们先来看看初始化。

1. 存储的初始化

在`init/1`中，首先需要创建用于存放映射关系的ETS表。很简单：直接调用`ets:new/2`即可。

需要访问这张表的进程很多，因此应该将它标为public；同时，为了让进程能便捷地找到这张表，还应该将它标为named_table（具名表）。表名由TABLE_ID宏指定，在此该宏被定义为模块名（sc_store）。

要事优先

按惯例，初始化函数或启动函数应该摆在 API 部分的首位。（其他模块中的 start_link 和 init 函数同样享有这个待遇。）遵循这类惯例有助于提高模块的可读性。

访问ETS表的方法有两种。第一种，同时也是最常用的一种，是利用ets:new/2函数返回的表句柄。正如pid可以唯一标识一个进程，该句柄也可以唯一标识一张表。第二种方法是利用表名。ETS接口要求每张表都有一个名字；但必须先设置named_table才能用表名来访问表，此外，多张表可以共用一个表名。在此采用具名表的原因在于我们不希望库的用户去追踪表句柄——一旦这样做，你就必须将句柄传递给所有会用到sc_store的进程，而且sc_store的每个API调用都必须包含该句柄。

现在问题来了：该在哪儿调用sc_store:init/0呢？好好想想。其实无非两条路：要么在应用行为模式模块sc_app中调用，要么在根监督者sc_sup中调用。前面曾经讲过，监督者应当简短可靠，Erlang/OTP的设计原则之一就是尽量不要在监督者模块中插入应用代码。仅在顶层监督者的init/1函数中插入少量代码问题还不大，因为一旦出了什么乱子，整个应用都无法启动[①]。但从原则上讲，这种做法仍然不受欢迎；与其这么做，我们宁可将初始化代码放到应用行为模式文件中去。修改src/sc_app.erl中的start/2函数如下：

```
start(_StartType, _StartArgs) ->
    sc_store:init(),                          <— 存储初始化
    case sc_sup:start_link() of
        {ok, Pid} ->
            {ok, Pid};
        Other ->
            {error, Other}
    end.
```

只要加上这么一句，sc_store便可以在应用启动后的第一时间完成初始化。如果再延迟初始化的时机（例如放在顶层监督者启动之后），就有可能在某些地方出现试图访问某张尚不存在的ETS表的风险。

接下来要处理的就是映射关系的存储，解决了这个问题才能通过给定的键从映射关系中求得对应的pid。

2. 表项的创建和更新

代码清单6-6中的sc_store:insert/2以表名为标识，简单地通过调用ets:insert/2同时实现了新映射关系的插入和现有映射关系的更新。如2.14.2节所述，ets:insert/2的第二个参数必须是一个元组。默认情况下，表中所有元组的第一个元素被视作键，其余元素被视作载荷（个数任意）。按键进行查找时，与该键对应的整个元组都将被返回。ETS默认表现为一个集合（set）：同一时刻一个键只能与一个表项相对应，如果表中现有的某个表项与插入的新元组具有相同的

① 言下之意是这种做法不会导致运行时难以发现的问题。——译者注

键，那么旧表项将被新元组覆盖——这恰恰是你所需要的功能。

请注意该函数不会对数据类型做任何检查；它才不关心你插入的是pid还是别的什么东西。这些都是内部代码，它们对调用者持完全信任态度。最终执行插入操作的代码都得由你来编写。你应该在系统的边界处执行有效性检查，之后便再无必要了。如果某处潜藏着什么问题，就应该通过测试来发现它。这种思想可以让你的代码保持整洁，同时也更易于维护。

现在你已经可以插入数据了，下面我们还要再把它查出来。

3. 数据读取

对于集合型（默认类型）ETS表，由于内部采用哈希表实现，按键查询操作是个速度很快的常数时间操作。如2.14.2节所述，`ets:lookup/2`的返回值是一个列表，其中包含了表中含有该键的所有元组。我们使用的正是集合型表，因此结果中要么只有一个元组要么一个都没有，参见代码清单6-6中的`lookup/1`函数。

找到指定的键所对应的表项之后，先将ETS返回的值转换成更顺眼的`{ok, Pid}`再返回。如果表中没有这个键，则直接返回`{error, not_found}`。这也是一种封装：对于函数的调用者而言，底层基于ETS的实现既不可见也不关键，那么自然也没有理由将ETS的返回值直接暴露给外界。

6

剩下的唯一操作就是表项的删除了。虽然仅有一行代码，但我们有必要对该操作的实现作一些解释。

4. 利用模式匹配删除表项

现在的问题是我们更希望按值（进程标识符）而不是按键来删除表项。这样做可以大大简化6.4.1节（代码清单6-5）中`sc_element:terminate/2`函数的实现：进程退出时，它只需要轻描淡写地说一句："请把和我相关的表项从表中删除。"请注意，我们的代码在结构上保证了键与进程之间一定是一一映射关系。解决这个问题的方法有几种。你可以另行维护一张倒排表；先查出与指定pid对应的键，再根据这个键去删除表项，但这样一来代码量会翻倍，而且每次插入、删除都会引发两次写操作，此前则只需要一次。此外，你也可以遍历整张表，挨个儿检查每个键所对应的值，直到找到特定的pid为止，可这太慢了（但在删除操作的频率足够低时仍然可以接受）。然而ETS还有一个强大的机制，无须逐一检查所有表项，只需利用模式匹配进行全表搜索即可。这种方法本质上仍然是全表扫描，但速度却很快，因为整个扫描过程完全是用C实现的，极力避免了无谓的数据复制。

一般来说，代码清单6-6中`delete/1`函数的使用频率要低于`insert/2`操作，甚至比`lookup/1`更要低，如此看来当前这个实现也就够用了（今后如果有加速的必要可以再加上倒排表）。前文所述的扫描过程是通过`ets:match_delete/2`实现的，其中`{'_',Pid}`为匹配模式。篇幅所限，在此无法详述ETS的模式匹配。针对ETS匹配函数的完整解释超出了本书的范畴，这些函数的功能着实强大，我们建议你参阅Erlang/OTP文档来了解详情。

删除操作采用的模式是`{'_',Pid}`。该模式可以匹配所有第二个元素为指定进程ID的二元组。配合`ets:match_delete/2`函数一同使用该模式，便可以删除表中所有匹配的表项（表中最多只有一个符合条件的表项）。轻松搞定！

匹配模式

这些模式都以 Erlang 项式表示，其内容组成可分为 3 种：

- ❑ 普通的 Erlang 项式和已绑定的变量；
- ❑ 单引号内的下划线原子（`'_'`）。其含义与普通 Erlang 模式中的下划线相同——用作省略模式或通配符；
- ❑ 形如`'$<integer>'`的模式变量（如`$1`、`$2`、`$3`…）。

举个例子，元组`{erlang,number,1}`可由模式`{erlang,'_',1}`匹配。通配符`'_'`表示你不关心这个位置上有什么东西。该模式可以匹配所有第一个元素为 `erlang` 且最后一个元素为 1 的三元组。利用模式变量，你还可以选择性地提取与模式相匹配的元组中的值。例如，（对于上述的同一个元组）模式`{'$2','$1','_'}`将得到列表`[number, erlang]`，这个结果由元组中各字段的排列顺序决定，而且模式变量的返回次序总是与它们的编号保持一致。详情请参见 `ets:match/2` 的文档。

这样一来sc_store模块也完工了。用于处理键与pid间映射关系的所有操作，外加初始化，都已齐备。在实现过程中，你进行了抽象，对应用的其余部分屏蔽了底层的实现（也就是对ETS表的使用）。大功告成之前还剩下一件事儿：创建一套可用作应用前端的用户API。

6.4.3　打造应用层API模块

应用层API模块通常与应用同名。（请注意，第4章中的服务器应用无须API。）在此，你将为simple_cache应用建立一个名为simple_cache的API模块。该模块为缓存服务的终端用户提供了以下接口函数：

- ❑ `insert/2`——将键/值对存入缓存；
- ❑ `lookup/1`——按键查询值；
- ❑ `delete/1`——按键从缓存中删除键/值对。

这套API并未包含应用的启动、停止功能，相关功能将交由`application:start/1`等系统函数来处理，详细内容参见4.3节。代码清单6-7中罗列了上述函数，集成了本章业已创建的所有功能。

代码清单6-7　src/simple_cache.erl

```
-module(simple_cache).

-export([insert/2, lookup/1, delete/1]).

insert(Key, Value) ->
    case sc_store:lookup(Key) of                    ❶ 检查键是否已经存在
        {ok, Pid} ->
            sc_element:replace(Pid, Value);
        {error, _} ->
            {ok, Pid} = sc_element:create(Value),
            sc_store:insert(Key, Pid)
    end.
```

```
lookup(Key) ->
    try
        {ok, Pid} = sc_store:lookup(Key),          ❷ 获取与键相对应的pid
        {ok, Value} = sc_element:fetch(Pid),
        {ok, Value}
    catch
        _Class:_Exception ->
            {error, not_found}
    end.

delete(Key) ->
    case sc_store:lookup(Key) of
        {ok, Pid} ->                               ❸ 清理
            sc_element:delete(Pid);
        {error, _Reason} ->
            ok
    end.
```

以下是对这些API函数的详细介绍。

- simple_cache:insert/2——该函数负责将参数中的键/值对存入缓存。为此，你需要先调用sc_store:lookup/1以判断表中是否已经存有和该键相对应的表项❶。如果表项已存在，可以调用sc_element:replace/2函数用新值替换现有的存储元素。如果还没有与该键对应的表项，你就必须创建一个新的sc_element进程来存储键/值对的值，并将键与进程ID间的映射关系存入sc_store。请注意，目前为止你的API尚未提供设置淘汰时间的功能，仅支持长达一天的默认淘汰时间。要添加该功能也很简单，只需再增加一个参数即可（出于向下兼容性考虑，还应提供一个支持默认淘汰时间的版本）。
- simple_cache:lookup/1——查询功能的实现很直接：用sc_store:lookup/1按指定的键查找对应的pid；如果找得到，就向对应的sc_element进程查询它所持有的值❷。其他情况下（键不存在，或进程于应答前终止）都返回{error, not_found}。凡是遇到这种需要依次执行一连串动作，且任一步骤失败后都应返回相同结果的情况，就应该使用try表达式。（try/catch相关详情请参见2.8节。）
- simple_cache:delete/1——与insert/2函数一样，删除指定的键所对应的表项之前也需要调用sc_store:lookup/1查找该键是否已经存在。如果不存在，直接返回ok即可。否则，将删除操作委托给sc_element:delete/1❸，该函数将进一步调用sc_store:delete/1（参见代码清单6-5中实现的sc_element:terminate/2）。令sc_element自己清理自己的好处在于，无论元素出于怎样的原因被移出，缓存都可以确保清理操作的执行——尤其是淘汰时间过期，或是某个bug触发了sc_element的异常导致进程意外退出。

搞定！你的简易缓存终于打造完毕可以运作了。下面我们来尝试一下（强烈建议你亲自动手）。试运行之前，请按4.3节中的指示编译src目录下的所有模块，并将生成的.beam文件悉数放到ebin目录下。然后请按下述指示（在应用根目录下）运行Erlang，启动应用：

```
$ erl -pa ebin

Eshell V5.5.5  (abort with ^G)
1> application:start(simple_cache).
ok
2>
```

试一试你刚刚写好的3个simple_cache API函数。现在你应该可以将任意类型的键值对存入缓存。进行实验时，你可以将src/sc_element.erl（参见代码清单6-3）中的默认淘汰时间改短些，比如改成60秒，以便确认淘汰时间过期后表项确实会自行淘汰。（改完代码后不要忘记重新编译模块并替换ebin下的.beam文件。）

6.5 小结

本章你打造了一套缓存服务，尽管我们对整个实现过程作出了详尽的解释，但反观你所编写的这几行代码，你会发现其实很简单。这个缓存带有我们所期望的CRUD功能，完全驻留于内存，甚至还带有基于淘汰时间的旧数据项自动淘汰策略。本章活用了不少概念，把本书第一部分讲过的内容用了个遍。

从基本的设计开始，你首先搭建了应用框架和应用行为模式模块。接着又实现了顶层监督者sc_sup，该监督者采用的监督策略有别于本书第一部分曾展示过的策略：它的行为更类似于一个工作进程工厂。应用骨架就绪后，你又动手创建了用于存储元素值和处理淘汰时间的sc_element模块、用于处理键与pid间映射关系的sc_store模块，以及整合了各个模块的应用API模块simple_cache。

现在你应该已经了解了OTP应用的创建方法，也明白了应该如何封装协议、如何管理状态、如何将程序组织成OTP组件，以及如何编写简洁可读的模块。从现在开始，我们将进一步钻研Erlang/OTP中的各种技术，进一步完善你的缓存应用——就当前的程度而言，Erlware的开发者认为它还无法胜任站点加速的任务。从根本上说，它还没有达到上线标准：一旦发生什么线上问题，你会手足无措——我们连最基本的日志都没有。另外，监控和事件处理也还没有纳入考虑范畴，除非人工进行检测，否则你无从知晓缓存服务的死活。别着急，这些问题都将在下一章得到解决。

第 7 章

Erlang/OTP中的日志与事件处理

7

本章概要

- ❑ Erlang/OTP的日志功能和SASL应用
- ❑ 事件处理和gen_event行为模式
- ❑ 创建自定义事件流

在上一章中你搭建了一套短小精干的缓存应用。该应用支持键/值对的存储和快速查询，甚至还可以自动淘汰老旧数据项。多种进程、监督者和数据表相互配合，优雅而简洁地实现了这些功能。该应用不仅实现得干净利落，功能也毫不示弱：应用及监督者的启动、来来往往的工作进程、数据的存取、淘汰时间的超时机制，还有针对数据表的各种操作，一应俱全。但从应用用户的角度出发，这些都是发生在应用底层的活动，难以被用户感知。比如，要想统计最近一小时内处理了多少次数据插入请求，你就无能为力了。更要命的是，一旦发生了什么故障，你连查都没法儿查。

在这一章，我们将向你介绍事件处理的概念。系统中总会不断涌现出各种事件，为了处理这些事件，Erlang/OTP专门提供了一套框架。通过该框架你可以自行创建事件流，还可以往系统中挂接各种事件处理器来响应系统中产生的事件。这套框架是标准OTP日志系统的基础。我们将向你展示这套系统的使用方法，以及如何通过自定义事件处理器来调整这套系统的行为。我们还会向你展示如何创建自定义的应用级事件流，这种手法可以为你的用户打开一扇门，让他们得以在你的系统中挂接自己的逻辑。本章将涵盖以下主题：

- ❑ 日志系统；
- ❑ 事件处理以及如何在日志系统中挂接自定义处理器；
- ❑ 自定义事件的创建和处理。

现在请考虑一个问题："Simple Cache的内部出现故障时该怎么办？"抱着这个问题，我们来谈一谈OTP的日志系统。

7.1　Erlang/OTP 中的日志

Simple Cache真要是在运行过程中出了什么故障该怎么办呢？就当前的设计来看，除非服务整个崩溃，否则你可能根本无法察觉有任何异常。（比如，在我们的设计中，元素存储进程在退出时会自行进行清理，而由于工作进程被设置成temporary型，监督者此时是不会作出任何反应的。）

这套缓存服务不能再这样沉默下去了，Erlang/OTP自带的精良装备可以很好地解决这个问题。这些装备就是日志应用、SASL应用，以及由gen_event行为模式提供的事件处理框架。它们共同构成了一整套强大的用于与外界进行各种信息交换的工具。利用gen_event，你还可以让其他应用接入你的系统。

> **此 SASL 非彼 SASL**
>
> 你也许知道，在网络协议中有一种通用授权协议框架也叫 SASL。但 Erlang/OTP 中的 SASL 应用与此风马牛不相及（早在 RFC 2222 撰成之前它就叫这个名字了）。在此，它表示的是身为 Erlang/OTP 五大基础应用（erts、kernel、stdlib、sasl 和 compiler）之一的系统架构支撑库（System Architecture Support Libraries）。这五大应用是 Erlang/OTP 其他部分的基石。其中 SASL 应用由若干用于系统管理的重要服务构成。

在本章的后续部分我们还会回到SASL和gen_event的话题。现在，我们不妨先对日志系统做一个初步的了解。

7.1.1　日志概述

Log4j等日志系统（或是log4c、log4r等）你多半用过。在各种软件开发工具中，日志的地位十分显赫，几乎所有编程语言都提供了一套适用于该语言的、可视为事实标准的日志系统。日志系统一般都设有多个严重级别，用于区分日志的重要程度。日志级别通常可分为五个，分别是：critical（或severe）、error、warning、info和debug。确切名称可能依系统的不同而不同。这几个级别的含义都显而易见，但每个级别的使用场合却没那么容易区分，针对这个问题，我们简单做一个总结。

- critical或severe——表示系统遭遇了灾难性故障或者客户已无法访问系统，此时应立即采取人工措施。使用这个级别时应慎之又慎，只有那种有必要在凌晨三点把人从床上拖起来的紧急情况才用得上这个级别的日志。
- error——告知系统运维人员系统中出现了一些不良状况，但并不严重。例如，某个子系统崩溃后重启了一次，或是某个客户会话因数据错误而被中断。这类问题理应立即得到修复，不过拖到明天再处理也无伤大雅。不要滥用这个级别，否则人们会逐渐忽略这些消息。
- warn——告知运维人员系统中出现了某些潜在的负面问题，但暂时无害。你可以选择忽略这个级别的问题，或是暂时绕过留待日后修复，以免捅出别的什么篓子或是给系统平添不必要的负载。
- info——表示一条通告性消息，用于将某个事件的发生告知给运维人员。这类事件可能是

好消息，比如“备份任务完成”；也可能有那么点儿令人沮丧，比如“邮件发送失败，五分钟后重试”。这个级别可以随便用，但也别太离谱，以免把你的运维团队淹没在各种毫无用途的细节之中。
- debug——提供各种巨细信息。这个级别基本上是给身为开发者的你准备的，用于协助调试运行中的系统，（在一定限度内）debug日志总是越多越好。除非有人明确地要求，否则一般是不输出debug消息的。

大多数日志系统都允许运维人员根据具体情况设置一个最低严重级别。如果想要把控一切，就调到debug级别，在该级别下你可以看到所有类型的消息。如果希望看到除debug消息以外的所有消息，就调到info级别。如果仅希望在发现问题时得到通知，则可以调到warn级别，以此类推。

日志系统还提供了一些其他功能，比如调整输出格式，给日志消息添加时间戳等。暂且不去关注这些细节，我们先来看看Erlang/OTP的日志功能是怎样运作的。

7.1.2　Erlang/OTP内置的日志设施

日志是个极为常用的系统需求，Erlang/OTP的基本发行版便提供了日志功能支持。其主要功能由标准库中kernel应用的error_logger模块提供，供OTP行为模式使用的扩展日志功能则由SASL应用提供。它不仅提供了输出日志的方法，还给出了一套用于实现自定义日志和事件处理的通用框架。

OTP的默认日志格式比较古怪，常用的日志解析工具都拿它没办法。因此你必须权衡是否选用原生的日志系统，如果你的系统需要融入现有的非Erlang的基础架构，那么可能不应该选用其他的日志系统；如果是要基于OTP创建全新的系统，情况则相反。同时，你也应当明了在选择外部日志系统时需要做出哪些妥协。为了做到知己知彼，让我们先来看看日志系统的主要API。

7.1.3　标准日志函数

用于投递日志消息的标准API比较简单，但该API仅支持3个日志级别：error、warning和info。很快你便会发现，这根本算不上什么限制，因为报告类型和事件处理器都可以自行添加。不过对于初学者而言，还是先将就一下吧。日志投递相关的API函数可以在error_logger模块（位于kernel应用中）中找到。以下是最基本的几个函数：

```
error_logger:error_msg(Format) -> ok.
error_logger:error_msg(Format, Data) -> ok.

error_logger:warning_msg(Format) -> ok
error_logger:warning_msg(Format, Data) -> ok.

error_logger:info_msg(Format) -> ok.
error_logger:info_msg(Format, Data) -> ok.
```

这几个函数的接口与标准库函数`io:format/1`和`io:format/2`相同（参见2.5.1节）：第一个参数是格式字符串，其中可以包含由波浪号（~）开头的转义码，如~w，第二个参数则是与转义码对应的值的列表。

我们先来打印几条日志看看。使用下列方法你可以通过调用info_msg/1来输出简单字符串：

```
2> error_logger:info_msg("This is a message~n").

=INFO REPORT==== 4-Apr-2009::14:35:47 ===
This is a message
ok
```

这样便向日志记录器发送了一条info消息，日志记录器会对该消息进行格式化，并给它加上一个含有严重级别和时间戳的标题。默认情况下，日志消息是直接输出到终端的，你也可以对系统进行配置，将日志输出到文件或将日志关闭。（示例中的ok是函数调用返回给shell的返回值，并非消息的一部分。）

使用info_msg/2可以在日志消息中掺入数据：

```
3> error_logger:info_msg("This is an ~s message~n", ["info"]).

=INFO REPORT==== 4-Apr-2009::14:39:23 ===
This is an info message
ok
```

转义码~s用于在格式字符串中插入其他字符串。有关格式字符串的详情请参阅io:format/2的文档，你只需记住第二个参数一定是个项式列表就可以了。一般来说，列表中的元素应该与格式字符串中的格式指示符一一对应（表示换行符的~n除外）。

这几个函数的容错性要优于io:format(...)。即便格式规范有误，消息也能输出，而不至于引起崩溃。例如，如果第二个参数中的元素数目不对，你将得到以下报告：

```
4> error_logger:info_msg("This is an ~s message~n", ["info",
➥this_is_an_unused_atom]).

=INFO REPORT==== 4-Apr-2009::14:42:37 ===
ERROR: "This is an ~s message~n" - ["info", this_is_an_unused_atom]
ok
```

如此一来即便在写代码时不小心搞错了日志消息格式也不至于丢失潜在的关键信息。这个特性看似没什么大不了的，但却非常重要。

让我们来考察一些更为实用的日志消息吧：

```
5> error_logger:info_msg("Invalid reply ~p from ~s ~n", [<<"quux">>,
➥"stockholm"]).

=INFO REPORT==== 4-Apr-2009::14:53:06 ===
Invalid reply <<"quux">> from stockholm
ok
```

当然，实际系统中的数据总是通过变量传入的，不会像这样硬编码在代码中。请注意~p：这个格式指示符特别有用。如我们在2.5.1节所述，它可以将给定的值格式化成更易于人类阅读的格式——在处理包含字符数据的列表或二进制串时尤为有用。

除了info_msg，请再分别尝试一下error_msg和warning_msg，并注意其中的区别。你会发现warning消息和error消息实际上是一样的；这是因为，在默认情况下，warning被直接映射至error。（出于历史原因，实际上只存在info和error两种消息。在启动Erlang时给erl加上+Ww参数可以取消warning到error的映射。）

除此以外，还有一组更为复杂的新式API函数，它们允许你以更灵活的方式定制报告格式，并允许你赋予报告一个类型。在7.2.3节中我们将用这些函数来产生日志事件。现在，请参阅Erlang/OTP文档来了解它们的详细使用方法：

```
error_logger:error_report(Report) -> ok.
error_logger:error_report(Type, Report) -> ok.

error_logger:warning_report(Report) -> ok
error_logger:warning_report(Type, Report) -> ok.

error_logger:info_report(Report) -> ok.
error_logger:info_report(Type, Report) -> ok.
```

你已经对Erlang日志系统中最基本的`error_logger`功能有所了解了，接下来我们将讨论SASL应用，看看它给日志系统带来了些什么。

7.1.4　SASL与崩溃报告

为了让后续的学习更为具体，我们先来创建一个简短的`gen_server`，如代码清单7-1所示，它启动后唯一的任务就是运行一段时间然后自行关闭。

代码清单7-1　错误报告示例：die_please.erl

```
-module(die_please).

-behaviour(gen_server).

-export([start_link/0]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).
-define(SLEEP_TIME, (2*1000)).

-record(state, {}).

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
    {ok, #state{}, ?SLEEP_TIME}.                ❶ 设置服务器超时

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(timeout, State) ->                   ❷ 引发异常
    i_want_to_die = right_now,
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.
```

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

与之前几章中的情形类似，这个gen_server也同样简单明了。此处再次用到了服务器的超时功能❶，我们在init/1函数中设置了一个以毫秒为单位的超时。在这段时间内服务器如果没有收到任何请求（它也不应该会收到请求），handle_info/2将被调用，且第一个调用参数为原子timeout。（请回想一下，handle_info/2回调函数用于处理带外消息。）但该函数有些异乎寻常：你在这儿写下了一些明显会导致异常并致使进程退出的代码❷（这两个原子绝不可能匹配）。在此我们只是想试试SASL的日志功能，别担心。

1. 基本错误报告

编译模块时，编译器会对这个扎眼的错误发出警告，无视它就好。随后，你将看到编译结果{ok,...}，表示代码终究还是通过了编译：

```
$ erl
Eshell V5.7  (abort with ^G)
1> c("die_please.erl").
./die_please.erl:29: Warning: no clause will ever match
{ok,die_please}
```

（请在这个示例的源码所在的目录下重新启动一个Erlang shell。）模块编译完毕之后，用start_link/0函数启动它，如下：

```
2> die_please:start_link().
{ok,<0.40.0>}
3>
=ERROR REPORT==== 4-Apr-2009::15:18:25 ===
** Generic server die_please terminating
** Last message in was timeout
** When Server state == {state}
** Reason for termination ==
** {{badmatch,right_now},
    [{die_please,handle_info,2},
     {gen_server,handle_msg,5},
     {proc_lib,init_p_do_apply,3}]}
** exception error: no match of right hand side value right_now
     in function  die_please:handle_info/2
     in call from gen_server:handle_msg/5
     in call from proc_lib:init_p_do_apply/3
```

服务器进程顺利启动，2秒之后退出，在退出的同时还输出了一些较有实用价值的错误信息。接下来，我们来启动SASL，看看事情会有何改观。

2. 启动SASL

手工启动SASL应用，如下：

```
4> application:start(sasl).
ok
...
```

除了此处列出的ok以外，屏幕上还滚动着大段难以解读的文本。别担心，那些只是SASL输出的info日志消息。SASL应用提供的功能可不仅限于日志，屏幕上一扫而过的那些全都是各种服务启

动时输出的info消息。每个进程启动时，都会把当前状态信息打印至日志中（此处的日志就是终端）。这些内容相当琐碎，限于篇幅我们在这个例子中略去了日志输出。

标准日志消息——即使用7.1.3节中列出的各基本函数输出的消息——在任何Erlang系统中都可用。应用还可以使用自定义的报告类型，但除非为之添加对应的事件处理器，否则自定义报告类型会被系统忽略。SASL中就有这样一个处理器，用于监听由标准OTP行为模式发送的报告，报告发送的时机很多，包括监督者启动时、监督者重启子进程时、子进程意外退出时，以及gen_server等行为模式进程崩溃时。在启动SASL时，你会看到SASL主监督者还启动了若干工作进程。

让我们来看看启动SASL之后上一个示例的运行状况会发生些什么变化：

```
5> die_please:start_link().
{ok,<0.53.0>}
6>
=ERROR REPORT==== 4-Apr-2009::15:21:37 ===
** Generic server die_please terminating
** Last message in was timeout
** When Server state == {state}
** Reason for termination ==
** {{badmatch,right_now},
    [{die_please,handle_info,2},
     {gen_server,handle_msg,5},
     {proc_lib,init_p_do_apply,3}]}
6>
=CRASH REPORT==== 4-Apr-2009::15:21:37 ===
  crasher:
    initial call: die_please:init/1
    pid: <0.53.0>
    registered_name: die_please
    exception exit: {{badmatch,right_now},
                     [{die_please,handle_info,2},
                      {gen_server,handle_msg,5},
                      {proc_lib,init_p_do_apply,3}]}
      in function  gen_server:terminate/6
    ancestors: [<0.42.0>]
    messages: []
    links: [<0.42.0>]
    dictionary: []
    trap_exit: false
    status: running
    heap_size: 377
    stack_size: 24
    reductions: 132
  neighbours:
    neighbour: [{pid,<0.42.0>},
                  {registered_name,[]},
                  {initial_call,{erlang,apply,2}},
                  {current_function,{shell,eval_loop,3}},
                  {ancestors,[]},
                  {messages,[]},
                  {links,[<0.27.0>,<0.53.0>]},
```

```
                    {dictionary,[]},
                    {trap_exit,false},
                    {status,waiting},
                    {heap_size,1597},
                    {stack_size,6},
                    {reductions,3347}]
** exception error: no match of right hand side value right_now
     in function  die_please:handle_info/2
     in call from gen_server:handle_msg/5
     in call from proc_lib:init_p_do_apply/3
```

之前的错误报告照常发出，与此同时你还得到了一份来自SASL的崩溃报告，其中包含了大量有关故障进程的额外信息。在调试实际系统中的崩溃时这类信息的价值尤为凸显。

3. SASL不管用时怎么办

接下来请再看一个例子。我们来创建一个与gen_server无关的简单模块，看看会发生些什么。代码清单7-2中的代码的行为和上一个例子基本相同，只是更为简单，且没有使用OTP。

代码清单7-2　非OTP进程崩溃示例：die_please2.erl

```
-module(die_please2).

-export([go/0]).

-define(SLEEP_TIME, 2000).

go() ->
    %% just sleep for a while, then crash
    timer:sleep(?SLEEP_TIME),
    i_really_want_to_die = right_now.          ❶ 导致进程崩溃
```

编译运行这个模块，将会看到：

```
6> c("die_please2.erl").
./die_please2.erl:10: Warning: no clause will ever match
{ok,die_please2}
6> spawn(fun die_please2:go/0).
<0.79.0>
7>
...
```

进程将于启动后2秒因badmatch错误❶而退出。即便启动了SASL，错误信息（未在此处展示）也远没有上一个例子中那么丰富。个中原因很简单：要启用SASL，还需要一些预备工作。当你以gen_server和supervisor等行为模式为基础构建应用时，这些工作可以由系统自动完成。你自己编写的进程可就享受不到这个待遇了——要想不劳而获可是行不通的。

稍微做一些变通，你可以部分达到目的。让我们再试一次，但这次用proc_lib:spawn/1代替普通的spawn/1来启动进程：

```
7> proc_lib:spawn(fun die_please2:go/0).
<0.83.0>
8>
=CRASH REPORT==== 4-Apr-2009::15:34:45 ===
  crasher:
    initial call: die_please2:go/0
```

```
    pid: <0.83.0>
   registered_name: []
   exception error: no match of right hand side value right_now
     in function  die_please2:go/0
   ancestors: [<0.77.0>]
   messages: []
   links: []
   dictionary: []
   trap_exit: false
   status: running
   heap_size: 233
   stack_size: 24
   reductions: 72
 neighbours:
```

这下SASL崩溃报告也输出了。proc_lib模块是Erlang stdlib应用的一部分，利用它你可以按OTP的方式来启动进程，它会按照OTP的一些必要规范对进程进行设置。虽然（就目前而言）可能性不大，但如果你硬是要编写脱离现成的行为模式的进程，那么最好用proc_lib来启动进程。从长期来看这样做会更有利。

现在你已经掌握了Erlang/OTP中基本日志功能的使用方法，配合SASL和脱离SASL的使用场景我们也分别做了考察。在下一节中，我们将介绍事件处理系统的工作原理，同时还会介绍如何通过挂接自定义事件处理器来进一步操控日志。

7.2 用 gen_event 编写自定义事件处理器

可能你并不喜欢错误日志记录器的默认输出格式。它与所有其他系统所使用的格式确实有较大的差异。你所在的企业可能已经围绕自己的日志格式开发了大量工具，这些工具无法与Erlang的日志格式兼容。这时你该怎么办呢？还好，错误日志记录器允许你在日志系统中穿插自定义的逻辑并输出自定义的错误信息。

7.2.1 gen_event行为模式简介

日志功能是构筑在Erlang的事件处理框架之上的，而该框架又以gen_event行为模式为基础。该行为模式为事件处理器封装了简单易用的接口。要想进一步调整Erlang/OTP的日志框架，就得编写新的gen_event行为模式的实现模块，好在这个任务并不难。gen_event行为模式接口与gen_server的类似：其中包含你所熟悉的init、code_change和terminate回调函数，也包含handle_call和handle_info回调。（在参数和返回值方面二者之间存在一些细微差别，在参阅文档之前请不要擅自假设。）不过gen_event接口用handle_event/2取代了handle_cast/2，你大概猜到了，这儿正是你接收错误日志事件的地方。

gen_event和gen_server之间的一个重要区别在于当你启动新的gen_server容器时，你需要告诉它应该使用哪个回调模块（这样也就可以了）；但在启动gen_event容器（有时也被称作事件管理器）时，起初是无须任何回调模块的。相反，在容器完成初始化之后，可以动态添加

（或删除）一个或多个处理器。当事件被投递至事件管理器时，事件管理器会调用当前已注册的所有处理器模块来处理事件。二者之间的区别如图7-1所示。

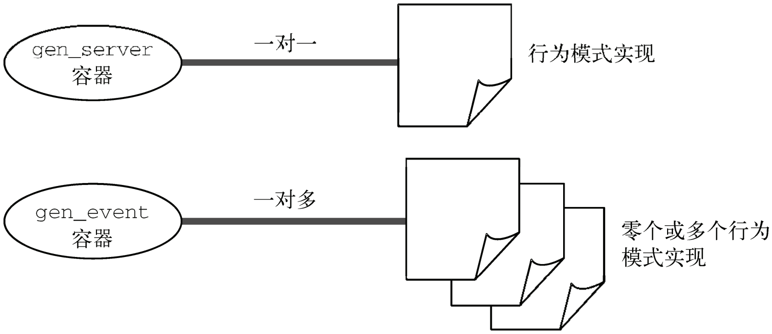


图7-1　gen_server和gen_event回调模块的使用。每个gen_server容器仅与一个特定的实现（回调）模块绑定，而每个gen_event容器可以动态增删任意多个回调模块

正是由于这种一对多的关系，在实现了gen_event行为模式的回调模块中一般是找不到start_link函数的；即便有，该函数一般也都会先检查容器进程是否已经启动（当然，仅在容器进程是经过注册的单例进程时这么做才有意义）。另外请记住，事件管理器要调用的回调模块可不止这一个，因此，不要对事件管理器进程的状态做出什么异乎寻常的举动，至于其他处理器，只能祈祷它们同样守规矩了。

和gen_server一样，为了便于访问，gen_event进程启动时也可以有一个注册名（正如第3章中的tr_server）。接下来，你将向注册名为error_logger的标准系统进程中添加一个处理器，该进程在所有Erlang/OTP系统中都存在。（这正是SASL启动时的工作。）当你调用error_logger模块中的日志函数时，所有日志事件都会被发送给这个进程。error_logger模块中还有一个专用于添加报告处理器的API函数，有了它你就无须关心事件处理器进程的定位问题了；该函数知道应该把处理器添加至哪个进程，并会连同该进程的注册名一起将调用委托给gen_event:add_handler/3。

7.2.2　事件处理器示例

你即将开发的简单日志事件处理器的骨架参见代码清单7-3。这只是错误日志记录器gen_event行为模式的一个最简单的实现。接收事件后它只会说：“OK，继续吧。”别的就什么也不会了。

代码清单7-3　自定义日志插件模块：custom_error_report.erl

```
-module(custom_error_report).

-behaviour(gen_event).

%% API
-export([register_with_logger/0]).

-export([init/1, handle_event/2, handle_call/2,
```

```
        handle_info/2, terminate/2, code_change/3]).

-record(state, {}).

register_with_logger() ->
    error_logger:add_report_handler(?MODULE).          <-- 将该模块添加为回调

init([]) ->
    {ok, #state{}}.

handle_event(_Event, State) ->
    {ok, State}.

handle_call(_Request, State) ->
    Reply = ok,
    {ok, Reply, State}.

handle_info(_Info, State) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

永不停歇

在早先的 gen_server 回调函数中，通过返回 stop，函数可以令服务器进程关闭。在 gen_event 进程中却不可以这么做（如果整个服务都被干掉，已注册的其他处理器可就无所适从了）。这种情况下，回调函数可以返回 remove_handler，收到该返回值之后 gen_event 进程会将处理器移除，并在最后时刻替你调用回调模块中的 terminate 函数。

现在你只需重新编译模块并调用模块中的API函数custom_error_report:register_with_logger()将它挂载到错误日志记录器的事件流中即可。

7.2.3 处理错误事件

接收到事件后就要进行处理。在当前这个例子中，我们仅需要将它们输出到屏幕。要恰当地展现这些事件，就必须了解它们的确切含义。error_logger中的函数会产生一组特定的事件。Erlang/OTP文档对此作了总结，参见表7-1。

表7-1 错误日志记录器事件

事件元组	来 源
{error, Gleader, {Pid,Format,Data}}	error_msg()
{error_report, Gleader, {Pid,Type,Report}}	error_report()
{warning_msg, Gleader, {Pid,Format,Data}}	warning_msg()
{warning_report, Gleader, {Pid,Type,Report}}	warning_report()
{info_msg, Gleader, {Pid,Format,Data}}	info_msg()
{info_report, Gleader, {Pid,Type,Report}}	info_report()

调用汇报函数时，如果没有指定类型，事件将被赋予默认的报告类型。其中由error_report、warning_report和info_report标记的事件，默认类型分别为std_error、std_warning和std_info。除了这3个类型之外，任意类型标识符都可用于用户自定义的报告类型。Gleader（进程组主管）字段暂且可以忽略，该字段用于指定标准输出的目的地。

代码清单7-4中的例子不太自然，请别在意，我们只是想用它展示一下应该如何利用上述内容去修订代码清单7-3中的handle_event/2回调。

代码清单7-4　处理error_logger事件

```
handle_event({error, _Gleader, {Pid,Format,Data}}, State) ->
    io:fwrite("ERROR <~p> ~s", [Pid, io_lib:format(Format, Data)]),
    {ok, State};
handle_event({error_report, _Gleader, {Pid, std_error, Report}}, State) ->
    io:fwrite("ERROR <~p> ~p", [Pid, Report]),
    {ok, State};
handle_event({error_report, _Gleader, {Pid, Type, Report}}, State) ->
    io:fwrite("ERROR <~p> ~p ~p", [Pid, Type, Report]),
    {ok, State};
handle_event({warning_msg, _Gleader, {Pid, Format, Data}}, State) ->
    io:fwrite("WARNING <~p> ~s", [Pid, io_lib:format(Format, Data)]),
    {ok, State};
handle_event({warning_report,_Gleader,{Pid,std_warning,Report}}, State) ->
    io:fwrite("WARNING <~p> ~p", [Pid, Report]),
    {ok, State};
handle_event({warning_report,_Gleader,{Pid, Type, Report}}, State) ->
    io:fwrite("WARNING <~p> ~p ~p", [Pid, Type, Report]),
    {ok, State};
handle_event({info_msg, _Gleader, {Pid, Format, Data}}, State) ->
    io:fwrite("INFO <~p> ~s", [Pid, io_lib:format(Format, Data)]),
    {ok, State};
handle_event({info_report, _Gleader, {Pid, std_info, Report}}, State) ->
    io:fwrite("INFO <~p> ~p", [Pid, Report]),
    {ok, State};
handle_event({info_report, _Gleader, {Pid, Type, Report}}, State) ->
    io:fwrite("INFO <~p> ~p ~p", [Pid, Type, Report]),
    {ok, State};

handle_event(_Event, State) ->
    {ok, State}.
```

这段代码的作用仅仅是将数据以略为不同的格式直接打印至标准输出，但通过它你应该能明白该如何编写自定义插件。请注意，有时你会收到一些无法与上述格式列表相匹配的事件，它们一般都是些可以忽略的系统消息，但你仍然需要在最后加上一个通配子句来处理这些事件，说上一句“OK”就够了。

建议你编译代码清单7-4中的代码，将它挂载到系统中，并按照7.1.3节中的日志示例发起几次调用，看看有什么动静。你也可以尝试一下error_logger模块中的自定义汇报函数，如info_report(Type,Report)（详情请参见Erlang/OTP文档）。

至此，我们已经完成了对Erlang日志基础功能中最重要的内容的介绍。通过这些内容，你应

该已经对下列内容的工作原理有了良好的认识：

- error_loggerAPI；
- SASL进程报告和崩溃报告；
- gen_event行为模式；
- 通过自行实现gen_event行为模式来自定义错误日志记录器。

现在，你可以把这些技术应用到上一章的Simple Cache应用中了。这正是下一节的内容，你将学会创建自定义的应用级事件流的方法。

7.3 为 Simple Cache 添加自定义事件流

如本章开头处所述，在Simple Cache系统上我们还有很多工作可做。在上一节中，我们介绍了如何在代码中添加标准错误日志消息，以及如何利用SASL应用去更好地了解OTP服务器以及监督者的行为。我们还介绍了事件处理系统的工作原理以及编写自定义事件处理器的方法。然而，抛开错误日志记录器，要想在自己的应用中创建应用相关的事件流又该怎么做呢？

对于这套缓存系统来说，你可以对外发布一系列的事件，用于向外界通告系统中发生的各种操作，如插入、删除、淘汰时间过期以及查询等。有了自己的事件流，你的用户便可以轻而易举地将事件处理器挂载进你的系统。而借助这些事件处理器，回答“在上一小时中缓存被查询了多少次？”，还有“缓存条目由于在淘汰时间内未被访问而被清理掉的频率有多高？”等问题也就轻而易举了。

在这一节中，我们将用gen_event行为模式创建一个自定义事件流。接着还要把它集成至Simple Cache应用中，为此我们需要将它挂入监督结构，并在各关键点处插入用于投递事件的代码。最后，我们还要编写一个事件处理器来截获这些事件。但首先，我们得先设计一下事件流的API。

7.3.1 事件流API

应用级事件系统的API由sc_event模块实现。和以往一样，你应该将实现细节尽可能地封装起来，仅留出一组易于使用的函数，供所有希望订阅该事件流的用户使用。这个模块很简单，参见代码清单7-5。

代码清单7-5 Simple Cache事件流API：src/sc_event.erl

```
-module(sc_event).

-export([start_link/0,
         add_handler/2,
         delete_handler/2,
         lookup/1,
         create/2,
         replace/2,
         delete/1]).

-define(SERVER, ?MODULE).
```

```
start_link() ->
    gen_event:start_link({local, ?SERVER}).

add_handler(Handler, Args) ->
    gen_event:add_handler(?SERVER, Handler, Args).

delete_handler(Handler, Args) ->
    gen_event:delete_handler(?SERVER, Handler, Args).

lookup(Key) ->
    gen_event:notify(?SERVER, {lookup, Key}).

create(Key, Value) ->
    gen_event:notify(?SERVER, {create, {Key, Value}}).

replace(Key, Value) ->
    gen_event:notify(?SERVER, {replace, {Key, Value}}).

delete(Key) ->
    gen_event:notify(?SERVER, {delete, Key}).
```

❶ 隐藏gen_event启动函数

❷ 隐藏gen_event处理器注册逻辑

❸ API函数

可以看到，这个API模块并未实现任何OTP行为模式。但它却提供了一个你所熟知的start_link()函数。在这个例子中，它的作用在于隐藏对gen_event:start_link/1函数的调用❶，该调用将启动一个新的gen_event容器进程并以该模块的名字在本地注册。

如我们在7.2.1节所说，gen_event行为模式的实现模块大都没有start_linkAPI函数。一般情况下，直接由监督者启动的都是gen_event容器（也叫事件管理器）而非gen_event实现模块，参见下列子进程规范示例：

```
{my_logger,
 {gen_event, start_link, [{local, my_logger}]},
 permanent, 1000, worker, [gen_event]}
```

（请拿代码清单4-3中init/1函数中的子进程规范与上述示例作一个比较。）启动之后，你便可以以my_logger为名来引用该容器进程并向其中添加处理器。

然而这些实现细节不应该暴露给其他代码。我们希望用户无须知晓管理器进程的进程名也可以添加事件处理器。为了达到这个目的，你不仅要提供start_link函数，还要为gen_event的标准注册函数add_handler/3及delete_handler/3提供一对包装函数❷，这与你在代码清单7-3中为error_logger的注册函数add_report_handler/1编写的包装函数类似。这样一来用户接口层就与进程注册名完全无关了。

紧随其后的4个函数才是真正的事件处理API。gen_event模块的notify/2函数可用于投递异步事件，类似于gen_server中的cast/2。API函数❸对协议进行了封装，与第3章（代码清单3-2）中tr_server封装服务器与客户端之间的协议的方法类似。但对于事件处理器而言，这层封装不如tr_server的封装来得彻底：你所添加的每个回调模块都必须能够解读API模块中定义的协议，因此你应该在文档中将协议写清楚（可能只是内部文档，视该事件系统的使用范围而定）。然而无论是哪一方，都不应该将协议中使用的项式暴露给其他部分的代码。表7-2总结了自定义事件流中的协议。

表7-2　Simple Cache中的应用层事件

事件元组	投　递　方
{lookup, Key}	sc_event:lookup/1
{create, {Key, Value}}	sc_event:create/2
{replace, {Key, Value}}	sc_event:replace/2
{delete, Key}	sc_event:delete/1

有了这套API，要想执行投递查询事件之类的事情，只需调用`sc_event:lookup(Key)`就可以了。修改事件协议或其他实现细节时，你也不必逐行遍历整个代码库去修改所有投递了该事件的代码了。

接下来，我们将把该模块挂接进系统，将事件系统与Simple Cache应用整合为同时启停的整体。

7.3.2　将处理器整合进Simple Cache

首先你应该明白，作为一个服务，sc_event模块启动的gen_event容器进程应该由监督者来管理。此外，每个OTP应用仅有一个根监督者，其他所有进程都由该监督者负责启动。现在的问题在于第6章中创建的根监督者采用的是`simple_one_for_one`型重启策略（参见6.3.4节）。这个重启策略可以很好地应对我们之前面临的问题，但这种监督者只能支持一种子进程类型；因此当前的gen_event进程无法直接挂到该监督者之下。

7

> **监督者的命名**
>
> 在<mod>模块中实现的服务，其监督者的实现模块用<mod>_sup命名再正常不过了。例如，sc_element_sup。

好在现有的监督者模块并不需要重写。（整个缓存服务的架构高度依赖于该模块。）不过你得重新给它起个名字：根监督者的角色不能再由它来承担了，你需要创建一个新的sc_sup模块来承担这个任务。首先，请将文件src/sc_sup.erl重命名为src/sc_element_sup.erl并修改文件中的模块声明，使之与新文件名相匹配。同时，记得更新sc_element.erl，将`create(Value, LeaseTime)`函数中调用的`sc_sup:start_child(...)`替换成`sc_element_sup:start_child(...)`。

更名为`sc_element_sup`的原监督者进程将和gen_event进程一起，变成新的顶层监督者的子进程。这个改动不难，只需写一个最基本的监督者模块，再添加两个子进程规范就行了。新的监督者模块如代码清单7-6所示。

代码清单7-6　新的根监督者：src/sc_sup.erl

```
-module(sc_sup).

-behaviour(supervisor).

%% API
-export([start_link/0]).

%% Supervisor callbacks
-export([init/1]).
```

```
-define(SERVER, ?MODULE).

start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

init([]) ->
    ElementSup = {sc_element_sup, {sc_element_sup, start_link, []},
                  permanent, 2000, supervisor, [sc_element]},

    EventManager = {sc_event, {sc_event, start_link, []},        <-  ❶ 新的sc_event进程
                   permanent, 2000, worker, [sc_event]},

    Children = [ElementSup, EventManager],
    RestartStrategy = {one_for_one, 4, 3600},      <- ❷ one_for_one监督者
    {ok, {RestartStrategy, Children}}.
```

可以看出，这个模块跟你在第4章中编写的tr_sup（代码清单4-3）很像。二者同为相对简单的one_for_one型监督者❷，都把自己的子进程视作相互隔离的不同类型的个体，在有必要重启子进程时，重启操作也是以单个子进程为单位独立进行的。这个新的监督者静态配备了两个子进程（各有各的子进程规范）：sc_element原有的监督者进程和新的sc_event进程❶。子进程本身是不是监督者，根监督者并不关心，但我们仍然把sc_element_sup和sc_event子进程分别标记为supervisor型和worker型，这样做可以辅助系统在某些特定情况下做出更准确地判断。（子进程规范中各字段的含义参见4.2.3节。）遵循这个模式，你可以在应用中以任意深度嵌套监督者，逐步谋求最佳粒度的监督结构。

现在，事件服务已经和应用融为一体，剩下的工作就是调整代码进而把事件抛出。也就是说，我们得回过头来从源码中找出合适的位置并加以修改，以便在恰当的时机将事件投递出来。

在这儿就不重复列出所有代码了，我们仅考察一个函数的改造过程。首先回顾代码清单6-7中的simple_cache:insert/1函数：

```
insert(Key, Value) ->
    case sc_store:lookup(Key) of
    {ok, Pid} ->
        sc_element:replace(Pid, Value);
    {error, _} ->
            {ok, Pid} = sc_element:create(Value),
            sc_store:insert(Key, Pid)
    end.
```

新建存储进程时应该投递一个create事件。为此，将代码修改如下：

```
insert(Key, Value) ->
    case sc_store:lookup(Key) of
    {ok, Pid} ->
        sc_element:replace(Pid, Value);
    {error, _} ->
            {ok, Pid} = sc_element:create(Value),
            sc_store:insert(Key, Pid),
            sc_event:create(Key, Value)          <- 投递进程创建事件
    end.
```

只需在恰当的位置插入一句sc_event:create()即可。一般来说，同一事件的投递点可以有多个选择，例如，在上个例中，你也可以从sc_element:create/1中进行投递。在此我们将

投递点选在了进程创建操作完成后的某处。表7-2中其余3个事件通知的添加方法与此类似，我们将之留作练习。请注意，调整lookup/1等函数时返回值也需要一并修改。

现在，应用已经改造完毕，它会不断地将自己的行为广播给所有的事件监听者。为了监听事件，监听者需要将相应的事件处理器插入你的事件流，与你在7.2节中订阅错误日志记录器时的操作类似。下一小节将展示如何处理这类自定义事件。

7.3.3 订阅自定义事件流

为了说明如何使用应用自定义的事件流（比如你刚实现的那个），我们将创建一个事件处理器并让它与Simple Cache的事件流对接。这与7.2.2节中错误日志记录器的事件处理器示例类似：二者都是gen_event行为模式的实现，它们之间的区别仅在于所处理事件的集合以及对具体事件的处理方法。

为了让这个示例有一些实际效果，我们直接将事件透传至错误日志中，这样也便于你观察。在实际应用场景中，你可以将这些事件转发至某个统计系统或远程监控系统。

代码清单7-7便是自定义缓存事件的日志处理器。该模块与代码清单7-3类似；简洁起见，我们省略了除handle_event/2以外的其他回调函数。

代码清单7-7 自定义事件处理器示例：src/sc_event_logger.erl

```
-module(sc_event_logger).

-behaviour(gen_event).

-export([add_handler/0, delete_handler/0]).

-export([init/1, handle_event/2, handle_call/2,
         handle_info/2, code_change/3, terminate/2]).

add_handler() ->
    sc_event:add_handler(?MODULE, []).

delete_handler() ->
    sc_event:delete_handler(?MODULE, []).

handle_event({create, {Key, Value}}, State) ->
    error_logger:info_msg("create(~w, ~w)~n", [Key, Value]),
    {ok, State};
handle_event({lookup, Key}, State) ->
    error_logger:info_msg("lookup(~w)~n", [Key]),
    {ok, State};
handle_event({delete, Key}, State) ->
    error_logger:info_msg("delete(~w)~n", [Key]),
    {ok, State};
handle_event({replace, {Key, Value}}, State) ->
    error_logger:info_msg("replace(~w, ~w)~n", [Key, Value]),
    {ok, State}.
```

❶ **add_handler/0和delete_handler/0函数**

❷ **error_logger消息**

add_handler/0和delete_handler/0❶用于简化日志功能的开启和关闭。处理器添加完毕后，经由sc_event投递过来的每条日志都会转换成相应的handle_event/2调用，直至你将

该处理器移除。

需要处理的事件的格式参见表7-2。你所需要做的就是按良好的格式把发生的事件以标准error_logger消息❷的形式打印出来。

我们建议你通过实验来验证上述的工作，步骤如下：改造Simple Cache应用、编写事件处理器、编译改造后的新模块，并按以前介绍的方法启动模块[①]。向缓存中存入一些数据并执行一些查询操作，随后，调用sc_event_logger:add _handler()添加日志处理器，再执行一些缓存操作，同时观察日志消息中是否出现了我们预料中的事件。移除处理器之后也再检查一下相应的日志是否会如期消失。另外再加一道附加题：请在缓存中添加一个于淘汰时间过期时投递的新事件，进行相应的修改并观察其工作情况。最后，记得让sc_app:start/2在sc_sup启动成功之后自动加上该事件日志处理器。

7.4　小结

对Erlang/OTP事件处理的学习就告一段落了，内容可真不少。一路学到这儿，你可能已经比很多在Erlang编程领域摸爬滚打了多年的人还要博学了！你学会了Erlang/OTP标准日志系统的使用方法，见识了怎样用gen_event行为模式来搭建日志系统（同时也学习了gen_event的工作原理），还学会了应该如何通过编写事件处理器来与日志流对接，进而自行调整日志输出。最终，你掌握了所有知识，借助这些知识，可以创建自己的应用级事件流，并使用自定义事件处理器将事件传递给错误日志记录器。

也许你和我们一样，对这一章的内容感到兴奋不已，甚至觉得有点儿喘不过气来。好好喘口气，舒缓一下情绪。下一章的内容可就更劲爆了（至少我们是这么认为的）。接下来我们将为你介绍Erlang的分布式机制及其运用方法。这是Erlang/OTP最为强大的能力之一，想必你已经迫不及待了吧。

① 启动模块的同时别忘了启动SASL。方法之一就是在用命令行启动erl时加上-boot start_sasl参数。（有关-boot参数的详情参见第10章）。——译者注

第 8 章

分布式Erlang/OTP简介

本章概要

- ❑ 分布式Erlang概要
- ❑ 运用Erlang节点和集群
- ❑ 实现一套简易的资源探测系统

在这一章里，我们不再给缓存应用添加任何新的功能了。为了给下一章的内容做准备，我们要先将目光转向Erlang的分布式编程能力。尽管与大多数其他语言相比，Erlang/OTP已经极大地简化了分布式编程，但分布式仍然是一个复杂的领域。你将在本章中学会运用分布式Erlang的基本技能，但要真正地掌握它则必须勤加练习。随着经验的积累，你会发现自己从传统串行编程中积累的经验在分布式场景下大都丧失了用武之地。不过别担心——不管怎样，这个过程都会充满乐趣！

8.1 Erlang 分布式基础

假设你在机器A和机器B上各跑着一个Simple Cache应用的实例。要是在机器A的缓存上插入一个键/值对之后，从机器B上也可以访问，那可就好了。显然，要达到这个目的，机器A必须以某种方式将相关信息告知给机器B。传递该信息的方式有很多，有些方式简单，有些方式复杂。但无论采用哪种方式，都涉及分布式，因为你需要进行跨机器通信。

Erlang极大地简化了某些类型的分布式编程，用不了几行代码，你瞬间就可以建立多台机器间的网络通信。这一切都以Erlang的两个基本特性为基础：

- ❑ 复制式进程通信；
- ❑ 位置透明性。

我们曾在第1章中对此做过简要介绍。下面我们将深入探讨一下这两个属性，看看它们是如何使分布式成为可能的。我们还会解释什么是Erlang节点，它们之间又是如何相互连接进而形成集群的。此外，我们还会向你介绍基本的安全模型、节点之间的通信方式，以及如何使用远程shell。最后，你将综合运用这些知识，实现一套复杂的分布式应用，该应用在后续章节中还会发挥作用。但是，首先，让我们来看一下为何Erlang的通信模型能够与分布式编程配合得天衣无缝。

8.1.1　复制式进程间通信

概括一下我们在第1章中的观点，在解决两段并发执行的代码段之间的通信问题时，最常用的模式就是让这两段代码共享某块内存，前提是它们都在同一台机器上运行。该通信模型如图8-1所示。然而它有很多问题，其中之一是当你希望每段代码都运行在独立的机器上时，就必须换用一种完全不同的通信方式。代码中的很大一部分将被迫重写。

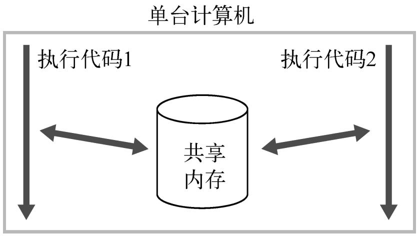


图8-1　传统的共享内存式进程间通信。这种方式要求通信进程双方都运行在同一台机器上，否则就不得不借助某种形式的分布式共享内存

这类问题正是Erlang的缔造者们从一开始就要解决的问题。要想在通信透明化的同时构建出容错的系统，要想让一台机器不至于因为相邻机器的崩溃或机器间的网络故障而宕机，就必须抛弃共享。

相反，Erlang的进程间通信采用的是严格的异步消息传递（发送消息后无须等待网络上的确认），接收方收到数据时实际上获取了数据的一份独立的副本；此后接收方将无法感知发送方对数据所做的任何操作，反之亦然。后续的任何通信都必须借助额外的消息才能进行。无论是运行在同一台机器上的进程（参见图8-2）还是运行在不同机器上并通过网络互联的进程（参见图8-3），这种模型都非常凑效。

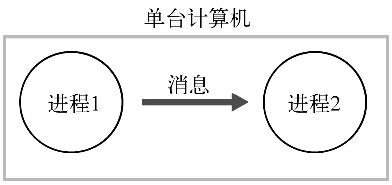


图8-2　单机上的Erlang进程通过消息传递进行通信。就效果而言接收方收到的总是数据的一份私有副本。在实际应用中，出于效率因素也可以基于只读共享内存来实现，但从进程的角度来看其实是一样的

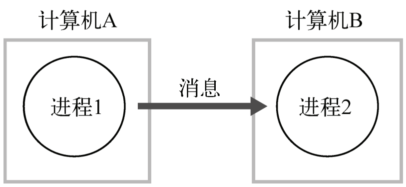


图8-3　不同机器上的进程通过消息传递进行通信。这种情况下显然需要将数据从一台机器复制到另一台上。除了由网络层引入的传输时延以外，和图8-2一般无二

可以看出，图8-2和图8-3之间的区别不大。程序在这两种模型中切换时，无须修改进程通信模式。

在Erlang中没有共享，只有消息传递，因此分布式还是单机本质上没有什么区别。大部分代码完全不用关心进程最终在何处运行。

然而在进行网络通信时仍然有很多需要注意的问题。在使用本地通信时，只要接收方进程还“活着”消息就一定能送达，而且几乎没有传输延迟。然而一旦涉足网络，就不得不考虑路由过程中的消息延迟以及网络本身的故障了。发送方通常无法辨别接收方到底是崩溃了还是因自身的bug而未能给出应答。出于健壮性考虑，即便是采用本地通信，发送方也应该对这些故障有所准备。但在分布式系统中总会存在多种导致不确定性行为的因素。

然而统一了网络和单机通信方式也不意味着你就可以在机器间随意迁移代码了。你好歹还得指定消息的目的地，尤其是要把消息发往哪台机器，对吧？

8.1.2 位置透明性

我们已经讲过，进程间的通信方式与接收方在本地机器上还是在远程机器上无关。这点在语法层面上仍然成立。以下是发送方向位于同一台机器上的接收方发送消息（本例中是个字符串）的语法：

```
Pid ! "my message"
```

然后再来看看怎么将同一条消息发向另一台机器：

```
Pid ! "my message"
```

抱歉抱歉，忍不住摆了你一道。没错，二者一模一样：`!`（发送）运算符具有*位置透明*的属性——接收方在哪台机器上并不重要，指引消息走向的信息统统都隐含在进程标识符之中了。Erlang会确保进程标识符在多机网络上的惟一性。这个属性使你可以无障碍地将程序的部署规模从一台机器扩展到数十台机器；你也可以反其道而行之，将那些原本部署在数十台机器上的程序集成到你的笔记本电脑上进行测试。

初看起来位置透明性好像没什么大不了的，然而你应该认识到，正是它大大释放了我们编程风格的自由度。当跨计算机通信的门槛不再那么陡峭，你终将获得翻越它的力量，并将之看作是习以为常的事情——除非是出于某些特殊原因，否则你的进程完全可以运行在相互独立的多台机器上，这时你便可以开始设计在以前看来复杂得超乎想象的系统了。

这两个属性——复制式通信和位置透明性——使Erlang分布式编程真正成为了一件令人愉悦的事情。在下一章中，这些内容都会被用到先前的缓存应用中去。但现在，你的缓存是不具备网络通信能力的。在另一台机器上另外启动一个缓存实例，两个实例完全无法感知对方的存在。为了给缓存应用添加分布式支持，首先就要改变这个局面：这些机器必须能够感知对方的存在。

8.2　节点与集群

先前我们故意模糊化了一个概念：我们所讨论的机器到底指的是什么呢？在很多场景下，一套硬件上只会运行一个Erlang VM（参见1.4节）；但有些时候——尤其是在测试和开发阶段，你需要在一台计算机上运行多个VM实例。按照目前启动`erl`（或`werl`）的方式运行起来的VM实例无从知晓也不关心其他实例的存在，因为它们之间并没有建立起网络连接。在单台计算机上运行多个基于Erlang的程序时（例如Yaws Web服务器和CouchDB数据库），这个模式是适用的。启用了网络功能之后，Erlang VM跑起来会更有意思。我们管这样的VM实例叫作节点。

一旦两个或两个以上的Erlang节点能够相互感知，我们就说它们形成了一个集群。（Erlang/OTP的官方文档通常称之为节点网络，我们不打算采用这个叫法，以免与计算机网络相混淆。）默认情况下，Erlang集群是一个全联通网络，如图8-4所示。换言之，集群中的每个节点都能够感知其他所有节点，任意两个节点都可以直接通信。

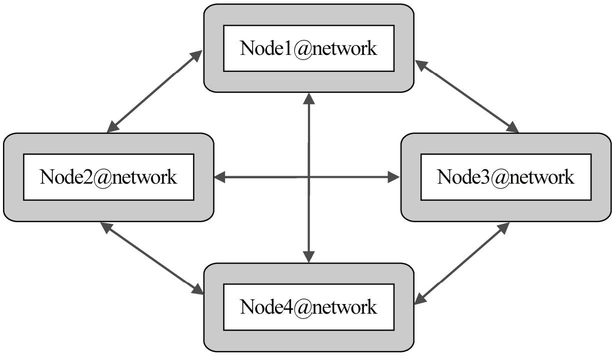


图8-4　经由网络形成集群的Erlang节点。集群中的每个节点都与其他所有节点直接相连：这是一个全联通网络

> **节点**
>
> 被配置成按分布式模式运行的 Erlang VM 就叫作节点。每个节点都有一个节点名，其他节点可以通过这个名字来找到该节点并与之通信。当前本地节点的节点名可以通过内置函数`node()`获取，节点名是个原子，格式为`nodename@hostname`。（不以分布式模式运行的 VM 的节点名恒为`nonode@nohost`。）在单台主机上可以同时运行多个节点。

8.2.1　节点的启动

只要给`erl`（或`werl`）加上命令行参数`-name`或`-sname`，就可以以分布式模式启动Erlang节点。第一种形式适用于配有DNS的普通网络环境，你需要给出节点的完全限定域名（fully qualified domain names）。例如：

```
erl -name simple_cache
```

第二种形式适用于完全限定域名不可用的情况；在某些生产环境下这种环境也是很常见的。还有的时候，举个例子，同处一个无线局域网内的两台计算机可以互联，但偏偏用DNS就走不通。碰到这些情况时，你就只能用短节点名了：

```
erl -sname simple_cache
```

只要所有节点同处一个子网，你就可以使用短节点名。

长短节点名不可混用

采用短节点名和长节点名的节点所处的通信模式是不同的，它们之间无法形成集群。只有采用相同模式的节点才能互联。

当Erlang VM以节点形态运行时，shell提示符中会包含节点名：

```
Eshell V5.6.2  (abort with ^G)
(simple_cache@mybox.home.net)1>
```

该节点名为`simple_cache@mybox.home.net`。可以看出它用的是长节点名（完全限定域名），它的启动参数为`-name simple_cache`。换用`-sname simple_cache`启动shell的话，节点名中主机部分带句点的内容就都没了，如下所示：

```
Eshell V5.6.2  (abort with ^G)
(simple_cache@mybox)1>
```

请在自己的机器上试验一下这两种模式。如果用的是Windows，请右击Erlang图标，选择属性，然后在目标命令（C:\...\werl.exe）中加上`name`或`sname`参数。你也可以在桌面上多复制几个图标再分别编辑它们的属性，这样一来只要点点鼠标你就可以用各种不同的节点名来启动Erlang节点了。 8

现在你已经知道该如何启动节点了，下一步自然就是让它们相互通信。

8.2.2 节点的互联

Erlang集群由两个或两个以上的节点组成。就数量而言，在同一集群内启动几十个节点没什么问题，但要跑上几百个的话就比较悬了。其原因在于维系机器之间的联络是需要一定的通信开销的，而Erlang集群又是一个全联通网络，这样一来这部分开销会随节点数的增加按平方规模增长。

隐形节点

借助一些特殊的节点，我们可以将多个集群合并成更大的、非全联通的集群。这类节点经过特殊的配置，不会对外传播其他节点的信息，它们甚至可以对其他节点隐身，以便对集群进行非侵入式监控。

一个节点不会主动搭理其他节点。你必须给它们一个呼朋唤友的理由；然而一旦探测到了别的节点，它便会持续追踪它们并与之交换已经和自己建立了连接的其他节点的信息，从而促成全

联通网络的形成。例如，假设节点A和B组成了一个集群，C和D组成了另一个集群，如果A和D相遇，它们就会互相交换B和C的信息，最终4个节点会共同形成一个更大的网络，正如图8-4所示。

不妨实际操练一下。请分别以a、b、c为节点名启动3个节点（每个节点都位于自己的窗口内）。然后按以下顺序连接这些节点：首先让a连接b，然后让b连接c。本例中的3个节点是在同一台机器上启动的，这倒不是说它们不可以独立运行在单独的机器上，但那样做就不得不牵涉我们尚未来得及介绍的和安全模型相关的内容。在考虑多台计算机的情况时，节点间的通信还可能会遭到防火墙的阻挠。简单起见，我们先让它们在一台机器上运行：

```
> erl -name a
Erlang (BEAM) emulator version 5.6.2 [source] [smp:2] [async-threads:0]
[kernel-poll:false]

Eshell V5.6.2  (abort with ^G)
(a@mybox.home.net)1>
```

请按同样的方式启动节点b和c。现在，在各个节点上调用内置函数nodes()，检查一下节点间的互联情况。目前它们返回的应该都是空表：

```
(b@mybox.home.net)1> nodes().
[]
```

下一步就是让它们互联。如果只是要建立连接，最简单的方法就是采用标准库函数net_adm:ping/1，如下所示：

```
(a@mybox.home.net)2> net_adm:ping('b@mybox.home.net').
pong
```

通信成功的话该调用将返回原子pong，否则就会返回原子pang。（看起来有点儿怪，在瑞典语中，pang就是拟声词“嘭”的意思，这就好像是在说“崩溃啦，嘭，执行失败”。）一切顺利的话，节点之间的连接应该就建立好了。你可以分别在3个节点上再次调用nodes()检查一下。在节点a上你应该能够看到b，反之亦然。在节点c上则仍然是张空表，因为它还没联络上其他两个节点：

```
(b@mybox.home.net)2> nodes().
['a@mybox.home.net']
```

如果在所有节点同处一台机器的情况下这一步仍然走不通，那么很可能是因为你采用了完全限定域名，却又没有配置好DNS。例如在接入了家庭局域网的某台PC上，可能有个名为'a@mypc.home.net'的节点，然而DNS又解析不出mypc.home.net这个地址。（可以用ping等常用命令行工具检查域名是否可用。）如果节点连不上，请用-sname代替-name再试试看。

接着，连接b和c，并再次调用nodes()：

```
(b@mybox.home.net)3> net_adm:ping('c@mybox.home.net').
pong
(b@mybox.home.net)4> nodes().
['a@mybox.home.net','c@mybox.home.net']
```

这个结果并不奇怪：b已经认识了a，现在又知道了c。在a和c上也运行一遍nodes()，你会看到整个集群已经是一个全联通的网络了：

```
(a@mybox.home.net)4> nodes().
['b@mybox.home.net','c@mybox.home.net']

(c@mybox.home.net)3> nodes().
['b@mybox.home.net','a@mybox.home.net']
```

最后再做一个实验，干掉节点b（比如直接输入q().；参见2.1.4节），然后看看在余下的节点a和c上会发生些什么：

```
(a@mybox.home.net)5> nodes().
['c@mybox.home.net']

(c@mybox.home.net)4> nodes().
['a@mybox.home.net']
```

虽然最初介绍这两个节点认识的节点已经不复存在了，但它们之间的联络却没有中断。这时如果重启b并令它与a或c相连，整个集群又会恢复为3个节点。

这一切是怎么运作起来的呢——尤其是当节点运行在多台独立的机器上时，情况又如何呢？

8.2.3　Erlang 节点如何定位其他节点并与之建立通信

检查一下系统中运行着的进程，看看其中有没有一个叫做EPMD的进程。在类UNIX操作系统中，你可以用ps：

```
$ ps ax | grep -i epmd
  758   ??  S  0:00.00 /usr/local/lib/erlang/erts-5.6.2/bin/epmd -daemon
```

EPMD代表Erlang端口映射守护进程（Erlang Port Mapper Daemon）。你每启动一个节点，它都会检查本地机器上是否运行着EPMD，如果没有，节点就会自行启动EPMD。EPMD会追踪在本地机器上运行的每个节点，并记录分配给它们的端口。当一台机器上的Erlang节点试图与某远程节点通信时，本地的EPMD就会联络远程机器上的EPMD（默认使用TCP/IP，端口为4369），询问在远程机器上有没有叫相应名字的节点。如果有，远程的EPMD就会回复一个端口号，通过该端口便可直接与远程节点通信。不过EPMD不会自动搜寻其他EPMD——只有在某个节点主动搜寻其他节点时通信才能建立。

> **更为高级的节点探测机制**
>
> 现有的系统允许你通过网络多播、广播等更为高明的手段去搜寻和连接 Erlang 节点。在 EC2 等可以随时按需增删服务器的云端环境下运行 Erlang 时[①]，你可能就用得上这些机制了。Nodefinder 就是这样一个项目，近期受到了不少关注，参见 http://code.google.com/p/nodefinder/。

请注意，Erlang默认的分布式模型基于这样一个假设，那就是集群中的所有节点都运行在一个受信网络内。如果这个假设不成立，或者其中的某些机器需要与外界通信，那么你就应该直接在TCP（或UDP、SCTP等）之上配合恰当的应用层协议来实现非受信网络上的通信。第3章中的RPC服务器采取的正是这种策略。此外，你还可以利用SSL、SSH或IPsec等技术建立加密隧道，

① 这意味着集群拓扑结构的变动会比较频繁。——译者注

甚至直接将Erlang的分布式通信层架设在SSL等传输协议之上（详情参见Erlang/OTP SSL库和ERTS用户手册）。

在典型的生产环境下，Erlang节点运行在受信网络内的多台机器上，其中的一个或多个节点通过Yaws、MochiWeb或标准库中的`inets httpd`等Erlang Web服务器与外界通信。在某些端口上你也可能会开放一些别的协议。除此以外，就别无其他途径可以从外界访问你的网络了。然而即便如此，一点儿安全防护工作都不做绝对是极为不明智的，哪怕仅仅是为了防范人为误操作呢。Erlang的分布式功能配备了一套基于magic cookie的认证系统，撇开防火墙因素，最常见的导致节点连接失败的原因就是cookie设置错误。在下一小节中，我们将介绍magic cookie的工作原理。

8.2.4　magic cookie 安全系统

找一台机器，在其上启动Erlang节点，并确保至少成功启动过一次，然后观察一下用户主目录。（在Windows上，该目录一般是C:/Documents and Settings/<username>或C:/Users/<username>，总之就是`%HOMEDRIVE%%HOMEPATH%`展开后的结果。）你应该会看到一个名为.erlang.cookie的文件。在文本编辑器中打开该文件，你会看到一个长长的字符串。它就是Erlang自动为你生成的cookie。在shell中可以用以下命令检查当前Erlang节点的cookie：

```
(b@mybox.home.net)1> auth:get_cookie().
'CUYHQMJEJEZLUETUOWFH'
```

返回的字符串应该与你在.erlang.cookie文件中看到的相同。Erlang节点只有在知晓其他节点的magic cookie的情况下才能与它们通信。节点在启动时会尝试读取.erlang.cookie文件，如果文件存在，它就会拿文件中的字符串当作自己的magic cookie。（修改该文件中的字符串并重启节点，重新执行上述命令，你就会看到修改后的字符串。）如果找不到，节点会新建一个cookie文件并写入一个随机字符串——这就是cookie文件最初的来历。试验一下：删除该文件并重启节点，你会发现该文件带着新的随机字符串又再次出现了。

默认情况下，每个节点都会假定所有与自己打交道的节点都拥有和自己一样的cookie。此前，当你在单台机器上（使用相同的用户账号和主目录）启动多个节点时，所有节点都共享同一个cookie文件，因此它们之间是可以相互通信的。要让运行于两台不同机器上的节点相互通信，最简单的办法就是将其中一台机器随机生成的cookie文件复制到另一台上，这样做一方面可以保证两个节点具有相同的cookie，一方面也可以保证该cookie不易被攻击者猜中。此外，除文件所有者以外的其他用户只应拥有该文件的读权限。

这个安全模型可以抵御一些简单攻击——例如，即便是在不受防火墙保护的计算机上启动Erlang节点，攻击者也无法轻易猜中你的cookie；然而更为重要的是，它还可以有效防范人为误操作。假设网络中运行着两个独立的Erlang节点集群，同时出于某些原因你又不希望它们一不小心合并成单个全联通的集群（比如这两个集群之间存在带宽瓶颈）。只要设置不同的cookie，你就可以确保这两个集群中的成员不会意外地因`net_adm:ping(...)`等原因而互联。

连接多台计算机

找两台在网络上相互连接的装有Erlang的计算机（数量越多越好），然后重复8.2.2节中的步骤。首先确保两边的cookie文件就绪；然后在两台机器上各启动至少一个节点，再调用`net_adm:ping/1`，这样就可以完成互联了。

对于更为复杂的配置，你也可以借助内置函数`set_cookie(Node,Cookie)`通过编程手段来设置cookie。采用这种方式，节点可以用不同的cookie与不同的节点通信。原则上说，集群中每个节点的cookie都可以不同，但在实践中整个系统往往会共用一个cookie。

接下来，不妨让你的节点相互联络联络，看看在分布式环境下Erlang的消息传递机制都能有些什么作为。

8.2.5 互联节点间的消息传递

在第1章和第2章中，我们简单介绍了如何用`!`和`receive`来进行消息传递。在此后的章节中，你又体验了用`gen_server:cast(...)`等API函数发送消息的方法。现在我们来考察跨节点的消息传递。请启动节点`a`、`b`、`c`并按之前的步骤令它们互联。（值得兴奋的是，这一次每个节点都可以运行在单独的机器上了。）完整地跑一遍下面的示例，你便会知道在Erlang中分布式编程是多么简单。

我们首先要演示的是如何与远程节点上的注册进程通信。在节点`b`上输入以下命令（还记得吧，每个表达式的末尾都得加上一个句点，光在行尾敲回车是行不通的）：

```
(b@mybox.home.net)2> register(shell, self()).
true
(b@mybox.home.net)3> receive
(b@mybox.home.net)3>   {From, Msg} ->
(b@mybox.home.net)3>     From ! {self(), "thanks"},
(b@mybox.home.net)3>     io:format("Msg: ~p~n", [Msg])
(b@mybox.home.net)3> end.
```

在第一个提示符下，你本地节点上注册了shell进程，注册名为`shell`。（有关进程注册的详情请参见2.13.3节。）注册函数返回`true`以示成功。紧接着是一个`receive`表达式，敲完最后一行的句点，回车，shell没能像往常那样打印出下一个提示符——它正在执行`receive`，等待着能与模式`{From, Msg}`匹配的消息的到来。模式中的`From`应该是个进程标识符，可用于向消息发送方发送应答；`Msg`则可以是任意数据。一旦收到符合条件的消息，shell进程首先会将自身的进程标识符作为应答回复给发送方，紧接着打印出接收到的`Msg`。等这一切全部执行完毕之后，你才能继续在shell中输入表达式，所以现在可以暂且把它放到一边了。

接下来，请在节点`c`上重复上述步骤。至此，`b`和`c`都进入了阻塞状态，等待着消息的到来。现在请切换至节点`a`，并输入以下内容：

```
(a@mybox.home.net)2> lists:foreach(fun(Node) ->
(a@mybox.home.net)2>                   {shell, Node} ! {self(), "hello!"}
(a@mybox.home.net)2>                 end,
(a@mybox.home.net)2>                 nodes()).
```

上述表达式用高阶函数`lists:foreach/2`遍历了一张列表，该列表由`nodes()`返回，其中包含与当前节点建立了连接的所有节点，目前该列表中应包含b和c。在遍历的过程中，你向列表中每个节点上的一个特定的目标进程发送了一条消息`{self(), "hello!"}`，该目标进程由元组`{shell, Node}`表示。在2.13.3节我们未曾提及消息的这种目标进程表示法，它表示消息应该发送给指定节点上的注册名为shell 的注册进程。如果不这么写，而只是写成`shell ! {...}`的话，消息将会被发送给本地节点a上的注册进程，这可不是你在此处想要的结果。

反观另外两个节点，它们应该各自收到了一条消息，并且遵照你的指示向发送方发送了应答并打印出了消息内容。这两个节点应该有如下表现：

```
Msg: "hello!"
ok
(b@mybox.home.net)4>
```

请注意，在前面的`receive`表达式中，你将发送方的pid绑定到了变量`From`上。不妨在shell中检查一下：

```
(b@mybox.home.net)4> From.
<5135.37.0>
```

这就是进程标识符的文本表述形式。由第一个数值可以看出，它指代的是一个位于外来节点上的进程——对于本地进程而言，pid中的这个数值一定是零。请注意，在用这个pid向发送方发送应答的时候，直接使用`From ! {...}`就可以了，无须指明进程所在的节点。目标节点的位置信息已经隐含在pid中了。（不用太在意pid中的各个具体数值——它们不过是一些临时分配的量，没有什么更深层次的含义。）在节点c上重复上述步骤，你会看到c上的`From`和b上的是一样的。

不过，那些应答消息都到哪儿去了呢？很简单——它们都在节点a的shell进程的信箱里排队呢。首先我们来看看shell进程的ID：

```
(a@mybox.home.net)3> self().
<0.37.0>
```

拿这个ID和其他节点上的`From`做个对比，你会发现它们中间的数值是相等的。不过这个ID的第一个数值为零，这表示该进程运行于本地节点上。现在，再来看看a上的信箱：

```
(a@mybox.home.net)4> receive R1 -> R1 end.
{<5316.37.0>,"thanks"}
(a@mybox.home.net)5> receive R2 -> R2 end.
{<5229.37.0>,"thanks"}
```

（记住，两个`receive`一定要使用不同的变量——否则第二个匹配会失败。）请注意，两个消息发送方的pid都是远程pid，它们的第一个数值都不为零。有意思的是，这两个pid与a上shell进程的pid在第二个数值上都相等。这是因为在每个节点的启动过程中，初始shell进程的启动时机基本上都是一样的。如果某个节点上的shell进程崩溃并重启，对应的数值会产生变化。你可以试试，在a上输入`1=2.`，然后再调用`self()`，就可以看到pid变了：在Erlang中，连shell都是采用进程来进行容错的。（shell进程崩溃后，信箱中的内容会丢失，但变量绑定关系仍然会保留。）

如你所见，Erlang的分布式通信非常简单，后续也不会出现更为复杂的内容了。这些就是它的全部精髓所在，剩下的无非都是些锦上添花的东西罢了。在正式开始分布式编程之前我们还有

一个重头戏，那就是学习如何通过本地终端来远程控制其他节点。

8.2.6 使用远程 shell

远程访问shell的能力是Erlang位置透明性的一个绝佳例证。说到底，启动一个普通的Erlang shell，实际上就是启动了一个可以与终端窗口的输入输出流进行对话的Erlang进程。此间的通信也是借由消息传递实现的，shell进程并不关心与自己相连的终端是否和自己同处一个节点。因此，我们完全可以在远程节点上启动一个shell进程，令它与本地节点上的终端相连，然后在其中完成各种工作。

Erlang shell的任务控制接口直接可以支持远程shell功能。回顾一下2.1.4节的内容，在shell中键入Gtrl-G便可以进入以下提示符：

```
User switch command
 -->
```

请在节点a上执行该操作。在提示符下键入h或?，可以看到帮助文本：

```
c [nn]            - connect to job
i [nn]            - interrupt job
k [nn]            - kill job
j                 - list all jobs
s [shell]         - start local shell
r [node [shell]]  - start remote shell
q                 - quit erlang
? | h             - this message
```

我们曾在2.1.5节中解释过如何使用任务控制，但当时我们并没有对r命令进行介绍。现在你大概已经猜到了。我们将要在节点a上启动一个在节点b上运行的任务。节点之间事先甚至无须建立连接，只要你乐意，完全可以随意重启节点b。请在a上输入：

```
--> r 'b@mybox.home.net'
-->
```

r命令以单个节点名为参数，在与该节点名对应的节点上启动一个远程shell任务，其过程与用s命令启动一个新的本地任务类似。如果节点名中含有句点（本例中就是），请务必用上单引号。和使用s命令时一样，该命令不会立即见效。请用j命令检查一下当前正在运行的任务：

```
--> j
  1  {shell,start,[init]}
  2* {'b@mybox.home.net',shell,start,[]}
-->
```

任务1就是原有的本地shell，注意一下任务2：结果显示该任务运行在节点b上。接下来请输入c2连接该任务——不过，正如*标记所指示的那样，2号任务就是默认任务，因此直接输入c就可以了：

```
--> c
Eshell V5.6.5  (abort with ^G)
(b@mybox.home.net)1>
```

看一下提示符，你已经跑在节点b上了！这意味着无论b所处的机器是在隔壁房间还是远在地

球的另一端,你都可以像坐在节点b的终端前一样在b上执行任何命令。其中也包括各种维护工作，如手工终止或重启进程、代码编译和升级以及监视或调试等。功能绝对强大！但能力越大责任也越大，一不小心的话你也很可能让节点宕机。

退出远程 shell 时千万要小心

使用完远程 shell 打算退出时，你的手指可能不自觉地就敲出了 `q().`，停！千万别回车！这个命令是 `init:stop()`的简写，用于关闭执行该命令的节点：也就是你的远程节点。恐怕每个 Erlang 程序员都曾经被这块石头绊过。要想安全退出，请使用 Ctrl-G 和 Ctrl-C（或 Ctrl-Break），这两个组合键只对本地节点有效。键入 Ctrl-G 加 Q，或 Ctrl-C（在 Windows 上是 Ctrl-Break）加 A，都可以在不影响远程节点的情况下关闭本地节点。

一般情况下，你会在某台机器上保留一个长期运行的节点，时而对它进行一些维护。为此，你可以在本地机器上启动一个新的临时节点，然后通过该节点远程连接至目标节点。维护工作完成后，一般就没有必要保留临时节点了，连按两次Ctrl-C即可退出（这说的是类UNIX系统，在Windows上可以用Ctrl-Break加A）。这种做法可以强制终止临时节点，同时结束远程任务。如果你还打算保留本地节点，则可以键入Ctrl-G，连接并返回原先的shell会话，随后你可以选择结束远程任务，也可以在二者之间来回切换。运行远程shell时的实际交互如图8-5所示。

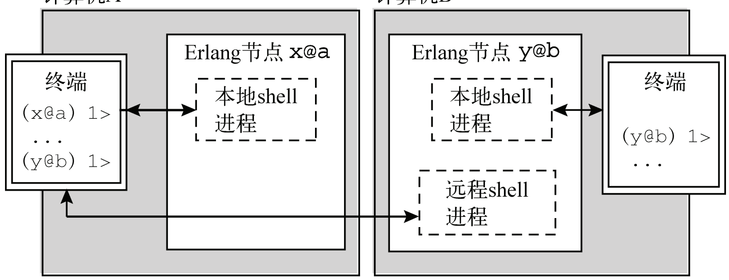


图8-5　远程shell的工作原理。远程shell虽然运行在计算机B上，但它与计算机A上的本地shell进程一样，是与A上的终端相连的。多亏了Erlang的位置透明性，这两种情况几乎没有什么区别

现在，你已经在分布式Erlang方面打下了坚实的基础。我们介绍了Erlang的分布式模型和节点的启动、互联方式，了解了安全模型和cookie的设置方法，学会了在节点间发送消息，还掌握了远程shell。在后续的章节中，这些知识将助你一臂之力，协助你将这些功能特性综合运用到你自己的系统中去。下一章我们将回到Simple Cache应用，不过在此之前，不妨再来找点儿乐子施展一下拳脚。

假设你在网络上跑着一个Erlang集群，在该集群上又运行着一大把功能各异的服务。当一个服务需要访问临近服务时，它如何才能找到目标服务呢？要是有一套资源探测系统，可以自动处理这些问题该多好。这样一来在集群内迁移服务时岂不就省事儿多了？

你可能在想："资源探测可没那么简单，至少也得写上10 000行代码才能搞定吧，而且还免不了要用上AF_INET、SOCK_DGRAM之类的晦涩的东西。"别担心——要知道你正在学的可是Erlang，不用花费多少力气就可以用一大把功能强劲的东西打出一套组合拳。

8.3　资源探测攻略

开发网络应用时最简单的一种做法就是将网络上各种资源的位置都硬编码到代码中。但这样做的话，一旦需要添加、迁移或移除各种资源，你就必须手工调整硬编码的配置信息（实施变更的原因可以有很多，包括服务扩展、重组、故障实例替换、代码升级等）。作为一名合格的软件工程师，你可能会将这些信息汇总到配置文件或数据库中。但你若是犯懒，直接将这些内容写进了代码，那就必须修改并重新编译相应的模块了。无论是哪种情况，这个手工调整的过程总是既低效又容易出错，而且随着资源的迁移和配置的变更还时常会把人搞得晕头转向。

与其采用硬编码，不如引入一套*资源探测*机制，让服务的供应方和需求方无须预先知晓系统的布局便可以相互定位。在这一节中，你将构建一个资源探测应用，其功能有点儿类似于黄页。集群中的每个节点都在本地运行一个该应用的实例。每个实例不断地探测和缓存集群中的各种有效资源。这种分布式的、动态的途径可以令系统更为灵活和强大，原因如下：

- *无单点*——这是一套点对点的系统；
- *无硬编码的网络拓扑*——你可以随处添加资源；
- *易于伸缩*——你可以随时加入更多资源；
- *可以在单个节点上运行多个服务*——整个探测过程是位置透明的，完全可以在单个节点上运行（该特性尤其适合于测试）；
- *易于升级*——可以动态完成旧服务的关闭和新服务的启动。移除的服务会被注销，新服务上线后很快便会被探测到。

资源一旦可以被动态探测，从开发到上线的全过程就都变得简单了（尤其是上线）。在着手实现之前，让我们先来澄清一些概念。

8.3.1　术语

为了明晰后续的讨论，我们需要先介绍一些概念。它们其实都是些非常通用的概念，只是不同的文献和实现各有各的叫法。资源探测，就是建立资源提供方和资源使用方之间的关系。为此，你需要同时跟踪资源提供方的供给和资源使用方的需求：换句话说，就是要将每个参与者的"供给"（I have）和"需求"（I want）整理成一张列表并加以维护。"供给"列表中的条目必须包含可直接使用和定位的具体的资源，而"需求"列表中的条目则只需要指明所需资源的类型就可以了（以便探测匹配的资源实例的发现）。表8-1对我们将要用到的各种术语做出了总结。

表8-1　资源探测相关的术语及其定义

术　语	定　义
资源	某个具体的资源（如fun函数）或指代某个具体资源的引用（如pid）
资源类型	资源的分类标签
资源元组	由类型标签和资源组成的二元组

下面让我们来详细分析一下这些概念。

1. 资源

所谓资源，就是某个特定的、具体的、可以直接使用的东西，比如一个fun函数或是一段二进制数据，也可以是用于指代某个具体资源的引用，比如pid、文件句柄、ETS表句柄等。一般来说，我们存储的都是系统中的资源的引用，而不是资源本身。

2. 资源类型

资源类型用于标识特定种类的资源。例如，在以资源的形式对外发布Simple Cache应用的一个实例时，可以将其标记成simple_cache类型的资源。在Erlang集群内，同一类型的资源可以有多个实例，无论采用什么方式实现，同一类的资源都应该具有相同的API。如果有谁声称自己在寻找simple_cache类型的资源，那么它便能查得集群中发布的这一类型的所有资源实例。

3. 资源元组

资源元组是由资源类型和资源共同构成的二元组。拿到了资源元组，就等于拿到了该资源。其中资源类型指明了资源的种类和访问方式。通过资源探测系统，你可以以资源元组的形式发布任何资源。任何人只要能够识别相应的类型标签，就可以定位并访问这些资源。

搞清楚了这些术语，就可以开始着手实现了。这套系统并不简单——分布式应用都不简单——不过你一定能行。首先我们要介绍的是其中的算法。

8.3.2　算法

假设你启动了两个相互连接的节点a和b（且二者已经完成了资源信息的同步），现在集群中又加入了第三个节点c。你所要解决的就是c与其他节点之间的资源信息同步问题。假设a和b各自持有一些x类型和y类型的本地资源实例（我们以x@a这种形式指代这些特定的实例），同时a和b还需要z类型的资源。（比如z可能是运行于a、b上的应用所需要访问的日志服务。）节点c持有一个z类型的本地资源，需要一个或多个x类型的资源，但不关注y类型的资源。

为了与其他节点保持同步，c上的资源探测服务器会向a和b发送消息，告知它们自己所持有的本地资源。节点a和b上的资源探测服务器收到这些消息后会缓存契合自己需求的资源z@c。随后，它们也会把自己的本地资源状况告知给c，c会缓存x类型的资源，并忽略y类型的资源。（这就好像是在玩“你说我就说！”的游戏一样。）期间的交互过程如图8-6所示。

在进入下一小节之前请务必仔细阅读上述内容。搞明白了这个算法，理解下面的实现就不难了。

接下来，我们就要来实现资源探测系统的主框架了。在读完本书之后，你会发现这套系统在很多地方都大有用武之地。

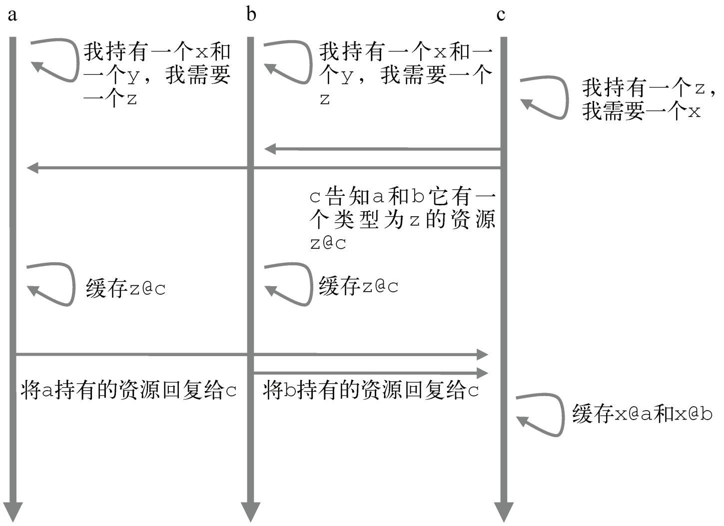


图8-6 资源探测算法。节点c正在加入集群；它持有a和b所需的z类型的资源，同时它在寻找x类型的资源

8.3.3 实现资源探测应用

整套系统的实现都以Erlang的消息传递为基础。实现手段当然不止一种，比如我们也可以采用网络多播或广播，但那就超出了本书的范畴了。为了保证该练习的简洁性和针对性，我们将省略监督和日志相关的功能，把所有代码都放到一个模块中。（在erlware.org上还有一个功能更为完备的版本，和你在前几章中开发的应用一样，那是一个真正的多模块的OTP应用。）

1. 模块首部

现在看来，这个模块毫无疑问应该用gen_server行为模式来实现。除了trade_resources/0等应用相关的API函数以外，下列的模块首部代码你应该很是眼熟了吧：

```
-module(resource_discovery).

-behaviour(gen_server).

-export([
         start_link/0,
         add_target_resource_type/1,
         add_local_resource/2,
         fetch_resources/1,
         trade_resources/0
    ]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).
```

在模块首部的末尾，定义了进程状态记录：

```
-record(state, {target_resource_types,
                local_resource_tuples,
                found_resource_tuples}).
```

此处一共定义了3个字段：target_resource_types就是上述的“需求”列表，这是一张你所需资源的类型的列表；local_resource_tuples则是“供给”列表，这张列表以资源元组的形式保存着本地节点上的所有资源；最后，found_resource_tuples用于缓存探测到的可以满足你的需求的资源实例（其中某些实例有可能就位于本地节点上）。

2. 服务器的启动和信息存储

接下来就该编写API函数了。首先就是start_link/0，该函数的实现可以直接照搬前面的章节：

```
start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
```

一如往常，为了让本地进程和远程进程都能够便捷地用进程名访问服务，你需要在本地注册服务器进程。写完start_link()，下一个自然就是启动过程中会用到的init/1回调了：

```
init([]) ->
    {ok, #state{target_resource_types = [],
                local_resource_tuples = dict:new(),
                found_resource_tuples = dict:new()}}.
```

此处定义了服务器的初始状态：target_resource_types字段被初始化为一张空表，found_resource_tuples和local_resource_tuples则由标准库中的dict模块分别初始化成空字典（即关联数组）。

到这一步为止，服务器启动相关的逻辑就完成了。现在再加两个API函数，它们的功能很类似，所以放在一起。这两个函数都会通过发送异步请求来将新数据写入服务器状态：

```
add_target_resource_type(Type) ->
    gen_server:cast(?SERVER, {add_target_resource_type, Type}).

add_local_resource(Type, Instance) ->
    gen_server:cast(?SERVER, {add_local_resource, {Type, Instance}}).
```

这两个函数分别用于向“需求”和“供给”列表中追加条目。第一个函数用于向“需求”列表中添加一个资源类型。第二个函数则用于添加位于本地节点上的资源实例，在添加的同时还会给该资源打上相应的类型标签。作为编程风格上的一个约定，两个函数发送的都是{Tag, Data}格式的二元组，其中Data本身可能也是个具有多个字段的元组，比如{Type, Instance}。三元组{Tag, Field1, Field2}当然也可以用，但出于一致性考虑，我们不打算混用不同长度的元组，而是统一用标签二元组作为协议消息格式。

这两个函数都用到了gen_server:cast/2，因此handle_cast/2回调也需要增加两个相应的子句来实现服务器端的功能。首先是add_target_resource_type：

```
handle_cast({add_target_resource_type, Type}, State) ->
    TargetTypes = State#state.target_resource_types,
    NewTargetTypes = [Type | lists:delete(Type, TargetTypes)],
    {noreply, State#state{target_resource_types = NewTargetTypes}};
```

首先，从服务器状态中提取当前所有目标资源的类型。然后，将接收到的资源类型加入当前的列表。为了避免重复，先执行一次删除操作。（如果列表中不包含待删除元素，lists:delete/2函数不会对列表产生影响。）

add_local_resource的实现类似：

```
handle_cast({add_local_resource, {Type, Instance}}, State) ->
    ResourceTuples = State#state.local_resource_tuples,
    NewResourceTuples = add_resource(Type, Instance, ResourceTuples),
    {noreply, State#state{local_resource_tuples = NewResourceTuples}};
```

先提取出所有本地资源，再将新的资源实例存放至指定类目下。这些操作由模块内的工具函数add_resourcec/3完成，该函数的实现如下：

```
add_resource(Type, Resource, ResourceTuples) ->
    case dict:find(Type, ResourceTuples) of
        {ok, ResourceList} ->
            NewList = [Resource | lists:delete(Resource, ResourceList)],
            dict:store(Type, NewList, ResourceTuples);
        error ->
            dict:store(Type, [Resource], ResourceTuples)
    end.
```

照例内部函数应该放置在模块的末尾。此处我们用标准库中的dict模块来维护资源类型与相应的资源实例列表之间的映射关系（这样一个类型就可以对应于多个资源实例），如图8-7所示。

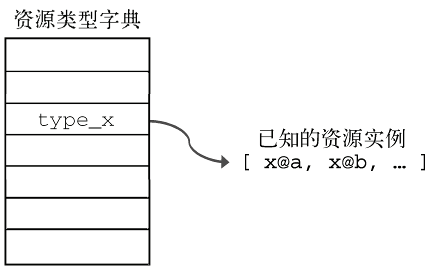


图8-7　维护资源类型与已知资源实例列表间映射关系的字典（关联数组）

如果键所对应的条目已经存在，就读出当前的值，在列表中追加资源后再写回；为了保证列表中资源的唯一性，事先也需要做一次删除。如果对应的条目不存在，则新建一张仅包含单个资源实例的列表。

3. 获取和交换信息

下一个API是fetch_resources/1：

```
fetch_resources(Type) ->
    gen_server:call(?SERVER, {fetch_resources, Type}).
```

该函数发起了一个同步调用，用于搜寻与某给定资源类型相符的所有资源实例。为了实现这个功能，你需要给handle_call/3回调添加一个对应的子句：

```
handle_call({fetch_resources, Type}, _From, State) ->
    {reply, dict:find(Type, State#state.found_resource_tuples), State};
```

此处的dict:find/2用于在当前的资源中查找Type。其结果要么是{ok,Value}要么是error，刚好可以用作fetch_resources/1的返回值，因此无须任何修饰——直接将结果回传就好。

最麻烦的工作由最后一个API函数完成：

```
trade_resources() ->
    gen_server:cast(?SERVER, trade_resources).
```

这个API函数通过一个简单的异步调用触发整个资源交换过程：该函数发送的原子trade_resources将由下面的handle_cast/2子句处理。该子句和下一个子句中的代码共同驱动了图8-6中的通信过程：

```
handle_cast(trade_resources, State) ->
    ResourceTuples = State#state.local_resource_tuples,
    AllNodes = [node() | nodes()],
    lists:foreach(
        fun(Node) ->
            gen_server:cast({?SERVER, Node},
                            {trade_resources, {node(), ResourceTuples}})
        end,
        AllNodes),
    {noreply, State};
```

这条trade_resources消息会促使本地节点上的资源探测服务器异步地向Erlang集群内所有互联节点上的资源探测服务器发起广播（也包括本地节点自身，这种对称性使你无须额外的代码便可以更新本地的资源匹配情况。）由于所有服务器进程在各自节点上都采用相同的注册名，广播的实现得以大大简化。

在服务器之间使用 cast

上述代码展示了两个不同的 gen_server 进程如何用 gen_server:cast/2 进行通信。由于是异步通信，消息投递完成后双方都会立即继续埋头处理手头的工作。一般情况下，不要用同步的 gen_server:call/3 来完成这类任务，gen_server:call/3 会阻塞服务器直至远程服务器给出应答——如果远程服务器刚好也正忙着调用你这边的服务器，双方就会陷入死锁。

这些广播消息的格式为{trade_resources, {ReplyTo, Resources}}，其中ReplyTo是发送方所处节点的节点名（由node()给出），Resources是一个数据结构（一个dict），内含发送方打算公布的所有资源元组。Erlang采用的是严格的复制式消息传递，因此你不用担心收到消息的进程搞乱你的本地数据结构。同时由于Erlang允许你在消息中发送任意数据——无须序列化——你可以直接把整个字典塞到消息中去。

收到这些广播消息后，节点将调用以下的handle_cast/2子句：

```
handle_cast({trade_resources, {ReplyTo, Remotes}},
            #state{local_resource_tuples = Locals,
                   target_resource_types = TargetTypes,
                   found_resource_tuples = OldFound} = State) ->
    FilteredRemotes = resources_for_types(TargetTypes, Remotes),    <-- 挑选出所需的资源
    NewFound = add_resources(FilteredRemotes, OldFound),            <-- 将之纳入已知资源列表
    case ReplyTo of
        noreply ->
            ok;
        _ ->
            gen_server:cast({?SERVER, ReplyTo},                     <-- 回复发送方
                            {trade_resources, {noreply, Locals}})
    end,
    {noreply, State#state{found_resource_tuples = NewFound}};
```

首先，你需要从当前的进程状态中提取出若干字段，这可以由子句首部的那段略显晦涩的模式来完成。请注意这个形如`#state{...}=State`的模式，这是一个别名模式：借助它你可以同时完成模式匹配和赋值。在阅读代码时，为了让注意力集中到数据的形态上，我们一般把变量名写在等号的右侧，但写成`State=#state{...}`也是可以的。

接着，就是检查有没有谁给你发来了你所需要的资源。与所需类型相符的那些资源随后会被添加至本地的已知资源列表中。最后，再给发送方一个应答就行了（发送方资源探测进程所处的节点由`ReplyTo`标识）。应答消息与广播消息的形态相同，只是将发送方节点替换成了原子`noreply`，以此表明该消息无须应答——否则这些消息将会在服务器之间反反复复地永远振荡下去。当最初发起广播的进程接收并处理完所有应答之后，它所持有的资源信息就与其他节点保持一致了。

8

以下是在前面用到过的几个内部工具函数。第一个函数只是简单地循环调用前面定义的`add_resource/2`函数，从而一次性添加多个资源元组。第二个函数要更复杂一些，它（通过`lists:foldl/3`）遍历传入的资源类型列表，为给定的每个类型构造了一张完整的已知资源的列表：

```
add_resources([{Type, Resource}|T], ResourceTuples) ->
    add_resources(T, add_resource(Type, Resource, ResourceTuples));
add_resources([], ResourceTuples) ->
    ResourceTuples.

resources_for_types(Types, ResourceTuples) ->
    Fun =
        fun(Type, Acc) ->
            case dict:find(Type, ResourceTuples) of
                {ok, List} ->
                    [{Type, Instance} || Instance <- List] ++ Acc;    <-- 创建一张二元组列
                error ->
                    Acc
            end
        end,
    lists:foldl(Fun, [], Types).
```

这段代码会找出与传入的每个资源类型相对应的资源列表，给其中的每个资源实例配上对应的资源类型，再将它们拼接成一个二元组列表（其中用到了列表速构，参见2.9节）。接着，该列

表又被合并入结果列表（即`Acc`）。（请注意，虽然这些列表不会太长，但为了避免平方复杂度，此处仍然选择从左侧开始构造列表，原因参见2.2.5节。）该函数的最终结果随后将被用作`add_resources/2`的参数。

4. 结语

一旦想明白，你就会发现这个算法还是相当直截了当的。即便做出了一些简化，它仍然可以满足大部分资源探测方面的需求。目前主要的欠缺在于我们无法自动触发资源探测。如果能在系统中的某些关键位置上加上相关逻辑（例如将之融入监督结构），我们就不用手工调用`trade_resources/0`了。这个问题将在下一章得到解决。

这是一套节点间的点对点协议，它仅依赖于异步消息，而且并不严格依赖于集群中所有其他节点的应答，因此整个系统的容错性相当好。可能发生的最坏情况就是节点因崩溃、重启或网络故障而突然消失，从而导致其他节点上的信息失真（你还没有加入自动清理消失节点上的资源的机制）。

有一个简单的集群稳定性改进方案，就是设立一个进程，周期性地试探目所触及的所有节点并触发资源交易，以此来克服偶发的联络中断问题或是可能导致节点间连接中断的崩溃。我们将该方案留给你作为练习。做到这一步之后，集群的健康度就相当有保障了。在以资源探测为基础设计系统时，你必须明确自己能从中获得哪些保障。在上述方案中，这主要取决于你对网络故障恢复时长的预期以及节点间自动重连的频率。

使用本章早先介绍的Erlang分布式编程技术，你构建了一套系统，其中的程序能够自行探测其他服务。如此一来就没必要硬编码集群内的网络拓扑和服务位置信息了。不到100行的代码，带你敲开了一扇门，构建高度动态、易于伸缩的服务从此不再神秘。在下一章，这些代码就要投入实战了。

8.4　小结

我们在本章介绍了Erlang分布式编程中的多个重要主题：

- 位置透明性和复制式通信；
- Erlang节点与集群；
- 基于cookie的访问控制；
- 远程shell的使用；
- 分布式技术综合运用：简单资源探测系统的实现。

你在本章所学的内容将给你的编程生涯带来无限多的可能性。我们真心希望本章中的这些短小精干的Erlang分布式编程实例能够激发你非凡的创造力。

在下一章中，你将把你所学到的知识——以及你所写的代码——用到Simple Cache应用中去，把它从单机服务改造成缓存集群，进而演变为一套会话存储服务，以便帮助Erlware的同仁为用户提供更棒的体验。

第 9 章 用Mnesia为cache增加分布式支持

本章概要

- ❑ 缓存分布式策略的选取
- ❑ Erlang内置数据库Mnesia简介
- ❑ 利用Mnesia将缓存分布至多个节点

在第7章中，你的缓存应用已然运转自如，第8章又把你带入了分布式Erlang的世界。你马上就得现学现卖咯。Erlware团队正打算给站点增加登录和会话功能：有了它，软件包的作者将可以进行多种软件包管理工作，包括版本更新、文档修订、软件成熟度标注等。你的任务就是进一步扩展缓存应用的功能，使之能够为该需求中的会话状态存储功能提供支持。

Erlware站点的最前端架设了一套无状态的负载均衡系统，因此页面加载请求可能会落到任意一台可用的Web服务器上。也就是说该Web应用中的所有服务器都要能够访问会话状态信息才行。问题在于，按照当前的设计，每个Web服务器都只能访问运行在本地的缓存，而且各个缓存实例对运行在其他Web服务器上的缓存实例的状况也一无所知。当前架构的简化视图如图9-1所示。 9

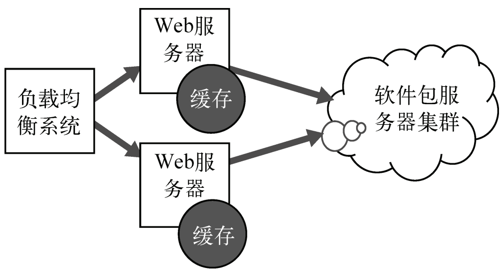


图9-1　缓存应用当前的简单架构。每个Web服务器只能访问运行在本地的缓存。受负载均衡的影响，请求会被派发至各个服务器，要想在缓存中存储会话状态数据，就必须实现分布式缓存

现阶段就把会话状态信息存入缓存的话，受负载均衡的影响，用户从一台Web服务器登录后，下一个页面请求很有可能会落到另一台Web服务器上。由于本地缓存中没有对应的会话状态，后一台服务器将无法识别用户。要是缓存实例之间的信息可以共享，那就万事大吉了。

9.1　分布式缓存

现在Erlware团队委托你来升级他们的缓存，以便在其中保存会话状态数据。升级后要求能够从任意一个缓存服务实例上读出与指定的会话密钥对应的会话状态；无论指定的是哪个密钥，也无论数据存放在何处，都要能读出来。像原先那样仅存储本地Web服务器上的数据肯定是不够用了，你需要的是一套能打通系统中的每个实例的分布式缓存。为此，必须先想清楚各实例间的信息交换方式。

9.1.1　选取通信策略

在设计分布式程序时，可供选择的通信方式主要有两种：异步通信和同步通信。在1.1.3节中我们曾对此做过简要讨论。采用异步通信时，发送方无须等待任何确认或应答。而在采用同步通信时，发送方会处于挂起状态，直至收到回复为止（即便只是“收到，多谢”之类的确认性回复）。Erlang消息传递的基本形式就是异步的，因为这种形式最为简单灵活：一般来说异步通信更适合分布式编程，而且同步通信总可以利用成对的异步请求/响应消息来模拟（`gen_server:call/3`就是这样做的）。

通信形式以异步为主，并不是说在构造程序时就不能采用别的范式。在这一节中，我们将针对通信策略的选取展开讨论，看看不同的通信策略会造就出怎样的系统，这些系统又分别会具备哪些属性。

1. 异步通信

异步通信有时也被称为“即发即忘”（fire and forget）[①]式或“即发即盼”（send and pray）式通信。消息一上路，发送方便撒手不管，继续干活。如果预期远端进程应该给出应答，发送方随后会伺机检查应答消息，如图9-2所示。一般来说，能否在指定时间内收到应答并不影响发送方后续的工作，至少部分工作不会受到影响。

异步通信的开销很低，是计算机系统间一种良好的基本通信形式。由于省去了各种检查、扫描、验证、计时等杂务，异步通信非常之快，尤其适用于创建简单而直观的系统。我们的建议是，除非万不得已，否则请尽量采用异步通信。

为了举例说明异步通信策略的适用场景，我们不妨与邮政业务作一个类比。譬如你正打算给祖母写信。写好信，把信塞入信封，贴上邮票，最后丢进邮筒。好了，处理完毕，接着写第二封信去吧。一般情况下信都可以寄到，当然也不一定；但无论如何，它都不会妨碍你寄信当天的后续活动。这种方式具有其固有的优点。信可能会寄丢，也可能过了很久才送到，祖母读完后也有

① fire and forget原本用于形容发射后可自动制导、无须人工干预的武器系统。——译者注

可能会一直捱到下个月才回信。不管怎么样，你在整个过程中是自由的，无须坐等结果。换言之，这套系统能够从容面对意外事件，只要系统中的各个角色能够正确处理这些意外，整个系统便可以持续运作。

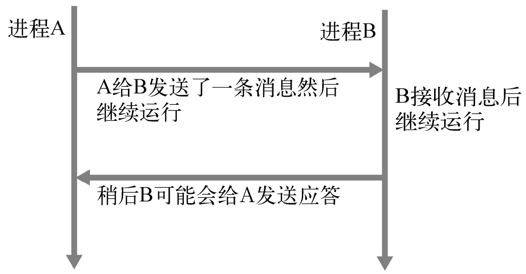


图9-2 即发即忘的异步通信：消息一发完，发送方便撒手不管，继续干活。应答消息将被单独发回

2. 同步通信

在同步通信中，每条消息都需要一个应答（哪怕只是一条用于确认消息送达的回执）。在收到应答之前发送方会被挂起，什么也做不了。由于发送方在等待应答的过程中处于阻塞状态，这种通信策略又被称做阻塞式通信 。典型的同步信息交换过程如图9-3所示。

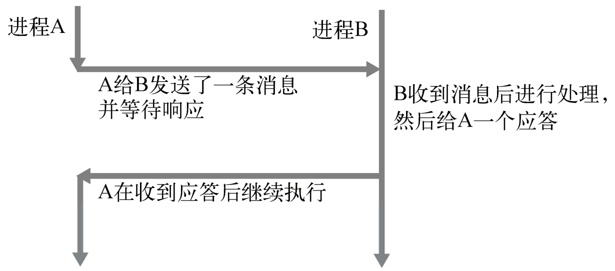


图9-3 同步阻塞式通信：在收到应答之前，发送方会被挂起。即使发送方的后续工作并不严格依赖于该应答，也会被迫中断，中断时长不短于消息往返一个来回所需的时间

同步通信最显著的缺陷就是在收到响应之前发送方什么都做不了（在分布式环境下，这段时间至少是网络中两台计算机之间消息传递时延的两倍）。另一方面，它的优势也很明显，那就是可以轻易地让系统在某一活动中保持同步。

比方说，一个生意人火急火燎地闯进政府办公室来给他的车交罚单。如果不结清罚款，再被发现违章停车的话他的车可就要被拖走了。他来到柜台前，递上一张支票，请办事员清空他的违章记录。和前面寄信的例子不同，这位老兄还不能走——他还没完事儿。他得在办公室里等着，直到柜台办事员处理完他的罚款，告诉他违章记录已经清空为止。等他拿到收据，确认系统已经进入预想状态，他才能离开办公室赶回去工作。

9

在实际应用中，同步通信需要与超时配合使用。政府交通管理办公室的那位老兄不会一直坐等到渴死为止，他会有一个最长等待时间的心理预期——一段时间之后，他必须放弃等待，要么认定缴款失败，要么碰碰运气假设缴款成功。如果他认为再等下去也是浪费时间（比如收据还没拿到，办事员又不在，而且看上去一整天都回不来），那么他可以先回去，择日再来，或者想想别的办法来解决问题。图9-4展示的就是带超时的同步调用。

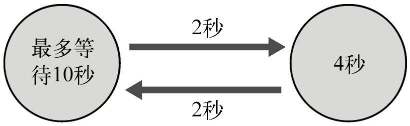


图9-4　带超时的同步通信。最长等待时间是10秒，整个信息交换过程耗时8秒：消息发送耗时2秒，请求处理耗时4秒，结果返回耗时2秒

前面两段有点儿跑题，主要是为了让你想清楚自己将要设计的缓存是一套怎样的系统：需不需要在某些时机同步某些特定的状态？向系统中插入或从系统中删除一段数据后，是否所有缓存实例都必须在第一时间反映最新的状态？也许你的系统并不需要如此严格的同步。或许它可以更异步一些，以删除操作为例，没有必要先汇总所有节点的执行结果再返回。这些设计决策对你的编码方案有着决定性的影响。在给缓存添加分布式支持之前，应该仔细考察一下各种可选方案，看看它们会对结果产生怎样的影响。

9.1.2　同步缓存和异步缓存

如前文所述，两种途径各有利弊。然而软件开发中的各种决策何尝不是如此。在这个案例中，选用同步方案还是异步方案必将对你的分布式缓存实现产生至关重要的影响。

1. 异步缓存

假设某人在站点上完成了登录，间隔一秒之后又针对另一个页面发起了请求，却被告知自己根本没有在该站点上登录过。要是可以容忍这种状况，那么缓存的插入操作就可以采用非阻塞通信来实现。别误会，不保证插入操作完成时系统状态的一致性并不意味着插入操作会频繁出错或是执行速度很慢——只是无法得到百分之百的一致性保障罢了。总体来看，服务的整体状态有可能会出现临时的不一致，而且在某个较低的概率下用户有可能会感知到这种不一致。

Erlang特别擅长实现基于异步消息的设计。在上一章中我们了解到Erlang节点可以组成集群。只要把所有缓存都纳入同一个集群，异步系统的设计就可以大大简化。第8章中的资源探测示例表明，在集群条件下可以很容易地向所有缓存实例发送广播。集群中的任意一个缓存实例在执行插入、删除操作时都可以向整个集群发起广播。

考虑以下登录事件序列：

(1) 用户登录网站；

(2) Web服务器创建会话；

(3) Web服务器调用`simple_cache:insert()`；

(4) 执行插入操作的函数向所有已知节点异步发送一条插入消息然后立即返回；

(5) Web服务器告知用户登录成功；

(6) 位于其他服务器上的缓存实例收到插入消息并进行处理。

该通信模式如图9-5所示。采用这种策略实现的缓存只能提供*弱一致性*保障：即便客户端已经被告知登录完成，服务器却并不能保证每个缓存节点都已经收到登录消息并完成了处理。

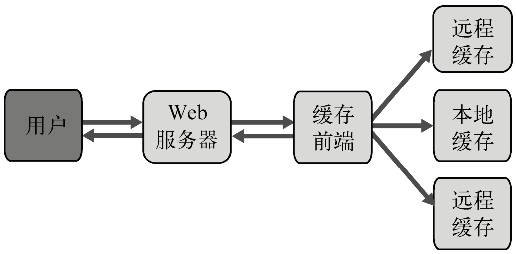


图9-5 异步缓存的交互过程：在执行写操作时，相应的函数会向所有缓存实例发送消息，消息一经发送，函数便会立即返回至调用方并告知操作完成，然而此时远程的缓存可能还没来得及收到消息并完成更新

简单得不能再简单了吧。假设缓存之间的通信能够抢在用户的下一次页面请求之前完成，那么一切天衣无缝。这个方案最大的优势就在于系统的运作方式简单直接（大道至简，简单总是好事），用不了几行代码便可以实现。

不幸的是，对于用户刚刚完成登录却又被告知"尚未登录"的情况，Erlware的负责人持零容忍态度——哪怕这种情况只是在系统负载高得离谱时才会出现。没办法，只能另寻出路了。

2. 同步缓存

要想在所有缓存实例都收到消息并如期插入数据之后再告知用户登录成功，则事件序列应调整如下：

(1) 用户登录网站；

(2) Web服务器创建会话；

(3) Web服务器调用`simple_cache:insert()`；

(4) 执行插入操作的函数阻塞，直至所有缓存实例完成数据插入操作；

(5) Web服务器告知用户登录成功。

该通信模式如图9-6所示。

同步模式提供的一致性保障与异步模式有所不同。在这种模式下，只有在所有缓存实例都拿到会话数据之后，系统才会告知用户登录成功。也就是说执行插入操作的函数必须先收到所有缓存实例的确认消息才能进行下一步动作。

这个模式可以采用好几种不同的手法来实现。第一种就是照搬上述描述，用gen_server:call/2逐个向各节点同步发送插入消息，直到最后一个节点更新完毕后才返回。然而当你有N个远程节点时，整个操作的耗时至少是消息在网络上往返一个来回的最小时延的N倍。这种程度的延迟理应极力避免。（但请注意，应答消息收集过程的耗时，是由反应速度最慢的缓存实例的响应时间

决定的——所有同步方案，无论怎么实现，都无法绕开这个限制。）

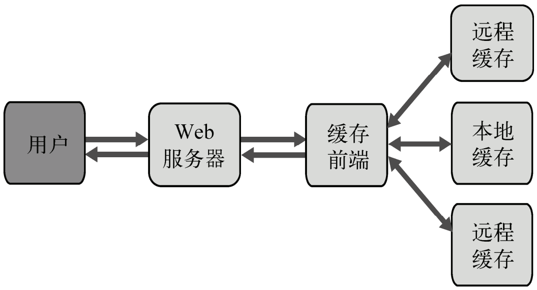


图9-6　同步缓存的交互过程：发起插入操作的缓存前端会一直阻塞直至所有缓存实例都完成更新为止。唯有那时Web服务器才会通知用户登录成功

高效实现分布式事务的另一途径就是研读诸如两阶段提交协议等内容。但那也未免又太小题大做了：照这么走下去，简直就是在开发分布式数据库。如果那就是你的目标，那么也许你应该先在Erlang/OTP库找一找，看看有没有现成的解决方案。

9.1.3　分布式表

回想一下第6章的图6-5，缓存应用在结构上由若干持有实际存储数据的存储元素进程组成，同时还有一张表，记录着每个键与对应的存储元素进程标识符之间的映射关系。需要在各缓存实例间以分布式方式存储的其实只有这张映射表而已。由于Erlang的位置透明性，没有必要在节点间复制数据：只要Web服务器之间的网络足够高效，即便存储进程位于另一台服务器上，调用远程存储进程的速度也要远快于调用原始的软件包服务器。因此，只要所有节点都能访问到键与pid间的映射关系，存储元素进程大可静静地待在创建它们的服务器上，免受迁移劳顿之苦。映射关系表、缓存实例以及存储元素进程之间的关系参见图9-7。

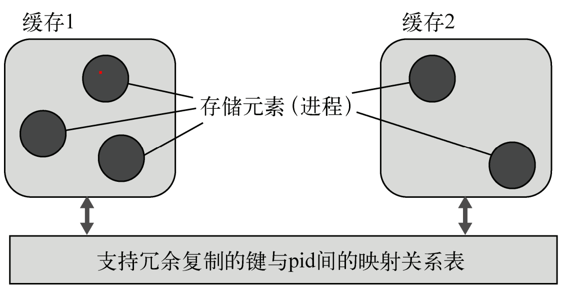


图9-7　两个缓存实例共享一张支持冗余复制的映射关系表，表中记录着键与进程标识符间的映射关系。Erlang的位置透明性确保我们无须关心进程所在的节点，从而简化了进程的访问方式；只有这张映射表需要在节点之间以分布式形式进行存储

现在来看，只要能够解决映射表的分布式存储问题就可以了。刚巧，Erlang/OTP为这个问题

提供了绝佳的解决方案：一套名为Mnesia的分布式数据库，看上去恰恰可以满足需求。了解到这些信息之后，你带着设计草案去找到Erlware的家伙们，他们很满意，放手去干吧。（不管做什么都得甲方点头才行啊！）

为了进一步完善这套方案，你还需要用上一章打造的资源探测系统来跟踪缓存节点，从而实现节点的动态增删。听起来工作量还真不小啊。赶紧开始吧，首先来讨论一下Mnesia，看看它是如何工作的。

9.2 用 Mnesia 实现分布式数据存储

Mnesia是一套轻量级的软实时分布式数据存储系统，支持冗余复制和事务，特别适合于存储离散的Erlang数据块，尤其擅长RAM中的数据存储。Mnesia天生支持Erlang，Erlang数据无须任何格式转换便可原封不动地存入其中。有鉴于此，它自然就成为了缓存应用首选的数据库方案。不过在正式启用Mnesia之前，有必要先了解一下Mnesia自身的一些局限。

就设计目标而言，Mnesia从来就没有打算取代SQL型数据库，也不应该用于管理分布于数十台机器上的数百吉比特的持久化数据。这种案例倒也不是没有，但我们并不推荐这种用法。Mnesia更适用于冗余数较低、尺寸较小的数据存储需求。对于大小适中的（基于磁盘的）持久化数据，或是需要跨进程共享的运行时数据，Mnesia都是不错的选择。要是出于容错和性能考虑，还需要将数据分布至多个节点，那Mnesia就更为擅长了。我们正打算解决的键与pid间映射关系数据的存储问题便是该应用场景的一个绝佳范例。

> **Mnesia 这个名字的由来**
>
> Mnesia 数据库刚刚诞生的时候，项目的研发负责人以其独特的幽默感给它取名为 Amnesia（失忆症）。然而很快管理层便通知他爱立信的产品线中绝不能容忍一个名为失忆症的数据库。于是这位负责人便削掉了那个不讨好的 A，剩下的就是 Mnesia 了。这个名字还不赖：在希腊语中 mnesia 就是“记忆”的意思。[①]

突然之间就引入了一套支持容错和冗余复制功能的数据存储系统，你可能还有点儿无所适从。不过没关系，实践出真知，你将在这一节建立一套切实可用的Mnesia数据库。在此过程中，你将充分掌握相关的基础知识，并在后续的9.3节中将之应用到缓存应用中去。

9.2.1 建立项目数据库

现在正式开始学习Mnesia。首先要建立一个数据库，用于存储Erlware软件包仓库的项目信息。当前这一版本的数据库只处理基本数据，用于存放用户信息以及用户所参与的项目的信息。这些信息将被分别放置在若干张表中。各数据间的关系模型参见图9-8。可以看到，在该模型中

① 也许是因为Erlang/OTP和Mnesia都出自瑞典，译者一直没有找到关于mnesia一词英文发音的官方说法。有鉴于mnesia源自amnesia（[æm'ni:zjə]），不妨读作['ni:zjə]。——译者注

User（用户）表和Project（项目）表各有两个字段，Contributor（参与人）表则维护着这两张表之间的关系。

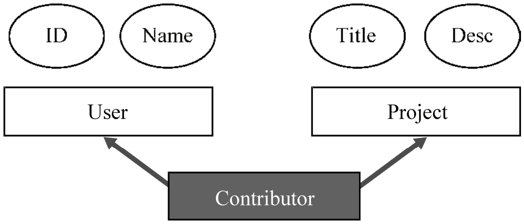


图9-8　项目数据库的数据模型。通过连接用户的ID和用户所参与的项目的名称（Title），Contributor表将User表和Project表关联了起来。其中用户ID和项目名称在数据库中都是唯一的

在Mnesia中，表项可由普通Erlang记录定义。代码清单9-1中罗列了该示例中用到的所有记录。

代码清单9-1　项目数据库的记录定义

```
-record(user, {
               id,
               name
              }).

-record(project, {
               title,
               description
              }).

-record(contributor, {
               user_id,
               title
              }).
```

过一会儿建表的时候，这些记录定义就会派上用场。但此前还有一些准备工作要做。建立数据库的过程可分为以下几个步骤：

- 初始化Mnesia；
- 启动节点；
- 建立数据库模式（schema）；
- 启动Mnesia；
- 建立数据库表；
- 向新建的表中录入数据；
- 对数据做一些基本查询。

先从Mnesia的初始化开始吧。

9.2.2 初始化数据库

在所有步骤之中，第一步就是Mnesia的初始化。该过程主要就是在磁盘上写入一些基本信息。首先启动一个Erlang节点，在启动时需要告诉它应该将Mnesia的相关信息写到文件系统中的什么地方。

1. 启动节点

在使用Mnesia时，请按如下方式启动Erlang节点：

```
erl -mnesia dir '"/tmp/mnesia_store"' -name mynode
```

上述命令会把数据的存放路径告知给Mnesia。（请注意，命令行中的单引号用于保留字符串两端的双引号。）同时，该命令通过-name选项让Erlang以分布式模式启动，以便启用Mnesia的冗余复制功能。（请按需选用-sname或-name选项。）节点启动之后，还需要在即将参与冗余复制的所有节点上建立一套空的初始数据库模式。

2. 建立数据库模式

所谓数据库模式（schema）①就是一些描述信息，其中记录着当前数据库中存有哪些表，表的详细情况又如何。一般来说不用关注它——它只是Mnesia用于跟踪自身数据的一种手段。当然，要想在多个节点上建立分布式数据库，就必须在所有节点上存放一份该模式的副本，以便让节点了解自己所存的数据的一般结构。为了防止Mnesia或整个Erlang节点关闭重启时丢失数据库信息，模式数据必须保存在磁盘上，存放路径由节点启动命令中的-mnesia dir "..."选项指定。（我们也可以让单个或多个节点，甚至所有节点仅在RAM中保存包括数据库模式在内的所有数据；但目前我们需要的是保存在磁盘上的持久化数据库。）

这个例子比较简单，只需要在本地节点上建立数据库模式即可：

```
(mynode@erlware.org)1> mnesia:create_schema([node()]).
```


该命令用于在本地节点上建立空数据库模式。如果执行失败，那么有可能是因为当前节点无法与列表中的某个节点建立通信，也可能是某个节点上已经有Mnesia在运行，亦或其中某个节点上残留有旧的数据库模式。（对于最后一种情况，可以调用mnesia:delete_schema(Nodes)来清理旧有模式——但请务必三思而后行：这样做会令当前数据库中的所有表都陷入万劫不复的境地。）

数据库模式就绪后，就可以启动Mnesia应用了。

3. 启动Mnesia

调用mnesia:start()便可手动启动Mnesia。Mnesia运行起来之后，可以调用mnesia:info()来核实数据库的基本信息，如数据库中现存多少张表，当前与多少个节点相连等：

```
(mynode@erlware.org)2> mnesia:start().
ok
(mynode@erlware.org)3> mnesia:info().
---> Processes holding locks <---
```

① 为避免与“模式匹配”的“模式”（pattern）混淆，本书将schema译作“数据库模式”，在上下文不存在混淆的情况下会简称为“模式”。——译者注

```
---> Processes waiting for locks <---
---> Participant transactions <---
---> Coordinator transactions <---
---> Uncertain transactions <---
---> Active tables <---
schema         : with 1        records occupying 422      words of mem
===> System info in version "4.4.8", debug level = none <===
opt_disc. Directory "/tmp/mnesia" is used.
use fallback at restart = false
running db nodes   = [mynode@erlware.org]
stopped db nodes   = []
master node tables = []
remote             = []
ram_copies         = []
disc_copies        = [schema]
disc_only_copies   = []
[{mynode@erlware.org,disc_copies}] = [schema]
2 transactions committed, 0 aborted, 0 restarted, 0 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
ok
```

用这种方法可以很方便地核实线上系统的配置，例如各节点间的全连通状况及一切是否配置完好等。

现在数据库系统已经初始化完毕，可以开始编写应用代码了，第一步是建表。

9.2.3　建表

建表操作完全可以直接在Erlang shell中进行，但由于shell对记录的支持很有限，这样做会有点儿别扭。作为替代，我们还是打算编写一个小模块来完成这项工作，参见代码清单9-2。和此前一样，为了节约篇幅我们略去了源码中的注释——在实际对外发布的代码中可不能这样。简洁起见，我们复制了代码清单9-1中定义的记录；通常应该将它们放入单独的头文件，然后让模块包含该头文件。

代码清单9-2　Mnesia建表模块

```
-module(create_tables).

-export([init_tables/0]).

-record(user, {
          id,
          name
         }).

-record(project, {
          title,
          description
         }).

-record(contributor, {
```

```
            user_id,
            project_title
           }).

init_tables() ->
    mnesia:create_table(user,
       [{attributes, record_info(fields, user)}]),
    mnesia:create_table(project,
       [{attributes, record_info(fields, project)}]),
    mnesia:create_table(contributor,
       [{type, bag}, {attributes, record_info(fields, contributor)}]).
```

可以看到，建表操作是调用函数mnesia:create_table(Name, Options)完成的，其中Options是一张{Name, Value}选项列表。在所有选项之中，最重要的一个就是attributes，几乎所有建表操作都会用上它。该选项用于指定表中所存记录的字段名。要是没有它，Mnesia会假定记录中仅有两个字段，分别名为key和val。实际应用中这种情况当然不多见，因此需要提供自定义字段名。不过，默认将第一个字段叫作key是有理由的：无论采用的是什么字段名，表的主键永远都是记录的第一个字段。

Mnesia 表与 Erlang 记录

对 Mnesia 而言，所谓的表无非就是一堆标记元组。而 Erlang 记录本质上就是标记元组（参见 2.11 节），但 Mnesia 无法感知表与同名-record(...)声明之间的关系。二者之间的联系需要由开发者来建立。（不强制关联所有同名的表和记录也是有好处的。有些时候，同名表与记录之间并没有联系。）

我们固然可以按{attributes, [title, description]}的格式直接将字段名写入代码，但最好还是用 record_info(fields, RecordName)来罗列字段名，以防今后修改记录声明。请注意，record_info/2 实际上并不是真正意义上的函数——它只在编译期有效（和记录语法中的#一样），在运行时或在 Erlang shell 中是无法调用它的。

9

在建表时我们仅设置了attributes选项，这意味着其余选项均采用默认值，具体来说：

- 表既可读也可写；
- 表仅驻留于RAM中（存储类型为ram_copies）；
- 表中存储的记录与表同名；
- 表的类型为set，即每个键最多只能对应一个表项；
- 加载优先级为0（最低）；
- local_content标记被置为false。

所有这些选项中，最重要、最需要搞明白的选项就是表类型和存储类型。下面我们首先解释一下什么是表类型，等建完表之后再来看看什么是存储类型。

1. Mnesia表的类型

我们曾在2.14.2节提过，ETS表可分为多种类型。Mnesia表也类似，只是候选类型略有不同：

Mnesia表可分为set型、ordered_set型和bag型3种。和ETS表一样，set型表中的键是唯一的，如果新插入的记录与现存的某个表项的主键相同，则新的记录会覆盖旧的记录。与此相反，bag型表可以容纳多个具有相同主键的记录，但这些记录至少要有一个字段的值不相等——同一条记录插入多次是没用的。

ordered_set型表与set型的行为相同，但set型表和bag型表都采用哈希表实现，ordered_set表则可以按主键的顺序保存记录。在需要按顺序遍历所有表项时，这一类型的表会很有用（哈希表无法保持具有实用价值的序）。

请注意，上述的Contributor表就是bag型的：

```
mnesia:create_table(contributor, [{type, bag}, ...])
```

也就是说这张表可以容纳多条具有相同用户ID的不同记录，进而可用于表示参与了多个项目的用户。

2. 表的存储类型

请编译该模块并执行init_tables()函数，然后再次调用mnesia:info()检查建表结果：

```
(mynode@erlware.org)4> c(create_tables).
{ok,create_tables}                                          ❶ 建表成功
(mynode@erlware.org)5> create_tables:init_tables().
{atomic,ok}
(mynode@erlware.org)6> mnesia:info().
---> Processes holding locks <---
---> Processes waiting for locks <---
---> Participant transactions <---
---> Coordinator transactions <---                     处于活动状态的表 ❷
---> Uncertain transactions <---
---> Active tables <---
contributor    : with 0     records occupying 312    words of mem
project        : with 0     records occupying 312    words of mem
user           : with 0     records occupying 312    words of mem
schema         : with 4     records occupying 752    words of mem
===> System info in version "4.4.8", debug level = none <===
opt_disc. Directory "/tmp/mnesia" is used.
use fallback at restart = false
running db nodes   = [mynode@erlware.org]
stopped db nodes   = []
master node tables = []
remote             = []                                    ❸ ram_copies 型表
ram_copies         = [contributor,project,user]
disc_copies        = [schema]
disc_only_copies   = []                                    ❹ disc_copies 型表
[{mynode@erlware.org,disc_copies}] = [schema]
[{mynode@erlware.org,ram_copies}] = [user,project,contributor]
5 transactions committed, 3 aborted, 0 restarted, 3 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
ok
```

可以看到init_tables()执行无误❶。mnesia:info()显示，现在有4张表处于活动状态❷，此前只有1张（数据库模式始终占用1张表）。还可以看到，应用中用到的表都是默认的

ram_copies类型❸。也就是说它们仅驻留于内存，这样做可以提升性能，但由于没有持久保存数据，一旦遭遇崩溃或重启数据就会丢失。

数据库模式表的类型为disc_copies❹，表示它会被写入磁盘，因而无惧重启；为了提高读取速度，这些表会被全部加载进内存。最后，❹之后的一行显示当前数据库中没有disc_only_copies型的表——表如其名，这些表仅存储在磁盘上，在访问速度上要比其他类型的表慢上许多。此外，目前disc_only_copies类型的表还不支持ordered_set。

不同节点上的表可以有不同的存储类型：比方说，可以将某张表配置成在一个节点上写磁盘，在其余节点上仅驻留于RAM。该配置甚至支持运行时修改，无须停机就可以在单个或多个节点上将一张表从RAM型存储切换成磁盘型存储，反之亦然。不过一般来说，应该在最初建表时就考虑好存储类型。

现在，所有的表都已经按正确的设置建立完毕，下一步就该向表中插入数据了。为此，还需要在create_tables模块中添加一些代码。

9.2.4　向表中录入数据

其他人在插入数据时是没有必要了解表的详情的，这些细节应该由API函数隐藏起来。添加API函数的同时也多出了一个校验机会，你可以在插入数据之前对数据进行一些一致性检查。比如说，新添加的用户至少要参与一个项目，并且不允许将用户加为尚不存在的项目的参与人。添加用户和项目的代码如代码清单9-3所示。

代码清单9-3　数据插入函数

```
insert_user(Id, Name, ProjectTitles) when ProjectTitles =/= [] ->
    User = #user{id = Id, name = Name},
    Fun = fun() ->
            mnesia:write(User),                                  ❶ 向表中写入用户记录
            lists:foreach(
              fun(Title) ->
                [#project{title = Title}] = mnesia:read(project, Title),
                mnesia:write(#contributor{user_id = Id,
                                          project_title = Title})
              end,                                               ❷ 插入参与人记录
              ProjectTitles)
          end,
    mnesia:transaction(Fun).                                     ❸ 设置事务

insert_project(Title, Description) ->
    mnesia:dirty_write(#project{title = Title,
                                description = Description}).
```

将上述代码加入create_tables.erl。记得从模块中导出函数insert_user/3和insert_project/2。

1. 事务

第一个函数有3个参数：新用户的唯一用户ID、用户的姓名，以及该用户参与的所有项目的

列表。请注意，执行用户插入操作的同时可能还有其他进程在访问数据库，这些操作必须相互隔离。因此应该以事务方式执行用户插入操作。Mnesia的事务具备通常所说的ACID性质。

- 原子性（Atomicity）——每个事务都是一个不可分割的单元，要么整体成功，要么整体失败。执行过程一旦出现失败，事务便会整体回滚，不会对数据库造成任何影响。
- 一致性（Consistency）——执行多个事务时，即便实际执行时间有所交叠，最终的效果应该像是各个事务按一定的顺序顺次执行一样，整个过程中数据库的状态始终保持完整和一致。
- 隔离性（Isolation）——每个事务都像是拥有一个自己的数据库副本一样，多个并发执行的事务不会相互干扰。在事务执行完毕之前任何人都观察不到事务的执行效果。
- 持久性（Durability）——事务执行成功后，它产生的变更便会生效。如果表保存在磁盘上，那么无论是重启还是崩溃都不会导致信息丢失。

在各种复杂操作的执行过程中，事务在保障数据库完整性方面起着至关重要的作用。Mnesia的事务非常简便——将你要做的工作写入一个（不含参数的）fun表达式，然后将之传给`mnesia:transaction/1`❸即可。在上述示例中，第一步是将用户记录写入User表❶。接着，用`lists:foreach/2`遍历传入的项目列表，对于其中的每个项目，首先检查该项目是否存在于Project表中（用期望的结果匹配`mnesia:read/2`的返回值），然后将相应的项目参与人记录写入Contributor表❷。上述操作中任一操作失败，都会导致整个事务回滚，Mnesia的状态也会恢复原样。

2. 脏操作

显然，在添加用户之前必须先想办法录入一些项目。这就是代码清单9-3中第二个函数的任务。这个函数走了一条捷径，使用了`mnesia:dirty_write/1`函数。以`dirty_`为前缀的Mnesia函数执行的都是脏操作，这些操作在执行时不会考虑事务或数据库锁。使用它们时必须格外小心。

一般而言，比起在事务中执行的普通数据库操作，脏操作要快得多。正确运用脏操作可以大大提升应用的执行速度。但是当心——没考虑清楚后果就滥用脏操作的话，很可能会导致数据不一致。一般来说脏读比脏写要安全；但不管怎么样，只要你心存疑虑，就请使用事务！（对于我们当前的应用场景来说，在应用的运行过程中突然插入一条项目记录并无大碍，哪怕该记录会覆盖原有记录也没有关系。）

3. 插入数据

现在该向表中录入数据了。请重新编译模块，然后在Erlang shell中执行下列命令：

```
(mynode@erlware.org)7> create_tables:insert_project(simple_cache, "a simple
        ➥cache application").
ok

(mynode@erlware.org)8> create_tables:insert_user(1,martin,[simple_cache]).
{atomic, ok}
```

上述命令会向对应的表中插入项目记录、用户记录、项目参与人记录各一条。执行完毕后数据库中的内容如图9-9所示。

Project

title	description
simple_cache	"a simple cache application"

User

id	name
1	martin

Contributor

user_id	project_title
1	simple_cache

图9-9　录入数据后的Mnesia表。User表和Contributor表采用数值类型的用户ID为主键，Project表则采用原子类型的项目名称为主键。每个Contributor表项分别引用一个User表项和一个Project表项

接下来，来看看如何从表中提取和查看数据，以便验证数据是不是真的已经完好地存入表中。

9.2.5　执行基本查询

在代码清单9-3中的`insert_user/3`中我们已经偷偷埋伏了一个`read`访问。由于使用了事务，可以直接调用`mnesia:read/2`函数来执行读操作。在不使用事务的情况下，可以使用脏操作来读取数据库，譬如在Erlang shell中可以这样做：

```
(mynode@erlware.org)9> mnesia:dirty_read(contributor, 1).
[{contributor, 1, simple_cache}]
```

无论`read`还是`dirty_read`，返回的都是与查询条件相匹配的记录的列表——在上述示例中，位于结果列表中的是Contributor表中所有主键为`1`的记录。如果找不到这样的记录，结果将是一张空表。此外，前面曾经提到Contributor表是一张`bag`型表。也就是说其中可以有不止一条和用户1相关的记录，这些记录全都会被纳入函数返回的结果列表。但对于一般的`set`型表而言，读操作返回的列表要么一个元素都没有，要么就只包含一个元素。

1. 使用带匹配规范的select

除按主键查询以外，还有一些更为灵活的查询操作。下面的例子展示的是如何用`mnesia:select/2`从`user`表中查询潜在的多条记录：

```
mnesia:transaction(
  fun() ->
    mnesia:select(user, [{#user{id = '$1', name = martin}, [], ['$1']}])
  end)
```

`select/2`的第一个参数是待查询的表，第二个参数是所谓的匹配规范（matching specification）。这些东西极其复杂，好在在简单情况下还算直观。每条匹配规范都是一个形如`{Head, Conditions, Results}`的三元组。`Head`是一个Erlang项式，用于描述查询模式，其

9

中的'$1'、'$2'等原子（注意它们外围的单引号）用于表示变量。在上述示例中，你所查询的是name字段为原子martin，id字段任意（'$1'）的#user记录。Conditions部分用于罗列作用于该匹配条件上的额外的约束条件，一般情况下都和这个例子一样留空。在最后的Results部分中，可以描述要从匹配到的每条记录中生成什么样的结果项式；此处可以使用'$1'等变量，它们最终会被替换成匹配结果中相应的值。

除了这些带数字编号的变量，还有一些具有特殊含义的原子：

- '_'（仅限于在Head部分使用）——无所谓，任意值都可以；
- '$_'（仅限于在Results和Conditions中使用）——与查询条件相匹配的整条记录；
- '$$'（仅限于在Results和Conditions中使用）——等价于依次罗列出在Head部分匹配的所有变量'$1'、'$2'、'$3'等。

事务执行成功后，返回结果的格式为{atomic,Data}，其中Data是事务中用于完成具体工作的fun表达式的实际执行结果；此处对应于select调用的执行结果——即一张包含匹配上的所有结果值的列表。在上述匹配规范中，Results部分为['$1']，因此匹配上的每条记录只会产生一个元素：即该记录中id字段的值。同时，由于整张表中只有一条记录的name字段为martin，因而mnesia:transaction/1调用的最终结果为：

```
{atomic, [1]}
```

对于有多个字段的表，我们往往只对匹配结果中的若干个字段感兴趣，这时就可以在Results部分采用[{'$1', '$2', '$3'}]或['$$']等规范来提取感兴趣的字段。有关匹配规范的详情请参阅Erlang/OTP文档中ERTS的用户手册。

2. 使用查询列表速构（QLC）

最后介绍一种表达能力更为强劲的Mnesia查询手段：查询列表速构（Query List Comprehension，QLC）。QLC是近期才加入Erlang/OTP的内容，其工作方式颇为奇特。从表面上看，它们与普通的列表速构很相似（参见2.9节），但却又必须嵌套在外观类似于函数调用的qlc:q(...)中使用。实际上外层的qlc:q(...)只是一个标记，用于让编译器区别对待其中的表达式。要启用该功能，模块源码中必须加上下面这一行：

```
-include_lib("stdlib/include/qlc.hrl").
```

（作为特例，在Erlang shell中可以直接使用qlc:q(...)，这是因为shell中没有文件包含的概念。）qlc:q(...)的返回值是一个查询句柄，配合qlc:eval(Handle)使用便可获取相应的查询结果。Erlang/OTP文档中的stdlib一节对QLC做了详细解释（参见qlc模块）。

QLC是一套通用查询接口，适用于ETS表、Mnesia表等各种具有表的特征的东西。通过实现相应的QLC适配器，甚至可以将QLC用在自定义的表结构上。在使用QLC之前，首先要用mnesia:table(TableName)函数建立一个Mnesia表句柄，该句柄将被用作QLC的输入参数。然后，就可以用普通的列表速构语法来实现各种过滤和聚合操作了。例如，要想实现早先示例中select函数所实现的功能，可以这样做：

```
mnesia:transaction(
  fun() ->
```

```
    Table = mnesia:table(user),
    QueryHandle = qlc:q([U#user.id || U <- Table, U#user.name =:= martin]),
    qlc:eval(QueryHandle)
  end)
```

相较于select和匹配规范，QLC是一套更为优雅的查询接口。就可读性而言上述代码比起之前的版本要清晰得多，代码的目的一目了然：首先从Mnesia的user表中找出所有U#user.name等于martin的用户记录；然后从其中的每个记录U中取出U#user.id，形成最终的结果列表。如果数据查询完毕之后还有别的事情要做，也可以选择在事务中使用QLC——任何可以在事务中使用的Mnesia函数都可以与QLC混合使用。

这一节只是对Mnesia的一个粗浅介绍——要想彻底说清楚Mnesia，那得写一本书才行。不过仅就用Mnesia搭建分布式缓存而言，这些内容应该是够用了，让我们继续吧。

9.3 基于 Mnesia 的分布式缓存

讨论完分布式缓存的概要设计，又学习了Mnesia的基础知识，现在总算可以开始开发了。要想让设计中的缓存正常运转，还有以下工作要做：

(1) 用Mnesia取代ETS；

(2) 让缓存能够识别出其他节点，从而进行必要的通信；

(3) 让缓存具备资源探测能力；

(4) 动态复制Mnesia表。

首先进行第一步，这样截至下一节末尾的时候，Mnesia就可以全盘取代ETS了。

9.3.1 用Mnesia取代ETS

9

还记得第6章中的sc_store模块吗？键与pid间的映射关系表就是由这个模块来封装的，它还向应用中的其余代码隐藏了存储相关的实现细节。现在，这层封装的意义终于凸显出来了，虽然数据存储层需要重新实现，但应用中其余部分的代码却一点儿也不用改。sc_store模块（代码清单6-6）中的关键函数有4个：

- init/0
- insert/2
- lookup/1
- delete/1

首先是用于设置ETS的init/0，现在你得用它来设置Mnesia表。我们后续还要进一步改造该函数，以便封装和冗余复制相关的逻辑；这些暂且放在一边，先把表建好再说。

1. 改写**init/0**

原先的init/0是这样的：

```
init() ->
    ets:new(?TABLE_ID, [public, named_table]),
    ok.
```

换用Mnesia后的新版本如下：

```
init() ->
    mnesia:start(),
    mnesia:create_table(key_to_pid,
                        [{index, [pid]},
                         {attributes, record_info(fields, key_to_pid)}]).
```

这是一张驻留在RAM中的普通set型表，键不可重复。和此前一样，record_info(fields, ...)会自动罗列出表项的所有属性（即字段）的名称。当然，还需要定义一个与这张表同名的key_to_pid记录。既然该应用中与存储相关的所有逻辑都在sc_store模块中实现，不妨就在该模块的起始处定义这个记录，如下：

```
-record(key_to_pid, {key, pid}).
```

请注意，在定义表结构时我们用到了{index, [pid]}选项，该选项用于设置索引。所谓索引，其实就是一些额外的表，用于加速非主键字段的查询。在创建索引时请务必牢记，索引会占用额外的空间。更重要的是，主表上的每一次写操作都会更新索引，这将导致启动速度和写入速度的下降。在使用索引时请务必做好利弊权衡。不过，在当前的应用场景下，为了快速查询与给定的pid对应的键，这些额外的代价是值得付出的（这正是我们引入索引的目的）。

启动设置修改完毕，下一步是改写insert函数。

2. 改写**insert/2**

第6章中的insert/2函数是这样的：

```
insert(Key, Pid) ->
    ets:insert(?TABLE_ID, {Key, Pid}).
```

对应的Mnesia版本也同样简单明了：

```
insert(Key, Pid) ->
    mnesia:dirty_write(#key_to_pid{key = Key, pid = Pid}).
```

请注意，在对表进行更新时我们用的是dirty_write。在这里Mnesia只负责最简单的键值存储，根本用不到事务。之所以选用Mnesia，主要是考虑到它的冗余复制功能，不过这是后话了。

接下来是lookup函数，也很简单。

3. 改写**lookup/1**

原ETS版本如下：

```
lookup(Key) ->
    case ets:lookup(?TABLE_ID, Key) of
        [{Key, Pid}] -> {ok, Pid};
        []           -> {error, not_found}
    end.
```

对应的Mnesia版本为：

```
lookup(Key) ->
    case mnesia:dirty_read(key_to_pid, Key) of
        [{key_to_pid, Key, Pid}] -> {ok, Pid};
        []                       -> {error, not_found}
    end.
```

由于是set型表，结果中最多只有一条记录，用dirty_read就够了。很简单吧。

但还有一个问题：在分布式环境下，你有可能会查到无效的pid。不妨考虑以下场景：假设有a、b两个节点，先在运行于节点a上的缓存中插入一些数据，然后到节点b上进行查询，这时是可以读取到正确的值的。现在强制终止节点a，再到b上进行查询，会发生什么呢？显然操作会失败。因为位于节点a上的存储进程皆已灰飞烟灭，而Mnesia数据库中的那些pid却仍然义无反顾地指向这些进程。综上，必须想个办法来清理那些包含无效pid的表项。

最简单的解决方案就是在执行查询操作时，对数据库中查到的pid做检查。以下函数用于判定给定的pid所指向的进程是否仍然存活：

```
is_pid_alive(Pid) when node(Pid) =:= node() ->
    is_process_alive(Pid);
is_pid_alive(Pid) ->
    lists:member(node(Pid), nodes()) andalso
    (rpc:call(node(Pid), erlang, is_process_alive, [Pid]) =:= true).
```

有了它，只需再在lookup函数中加上一个检查就可以了：

```
lookup(Key) ->
    case mnesia:dirty_read(key_to_pid, Key) of
        [{key_to_pid, Key, Pid}] ->
            case is_pid_alive(Pid) of
                true -> {ok, Pid};
                false -> {error, not_found}
            end;
        [] ->
            {error, not_found}
    end.
```

采用这个方案，失效的pid仍然会留存在Mnesia中，直到对应的表项被删除或被覆盖为止。不过没有关系，因为lookup函数会忽略无效的pid。请仔细考虑一下，要想高效地清理失效的pid该怎么做？为此需要改动哪些模块？

9

最后，要改写的函数就只剩下delete了。在用Mnesia替换ETS的过程中这是唯一的一处挑战。

4. 改写**delete/1**

此前的delete函数如下：

```
delete(Pid) ->
    ets:match_delete(?TABLE_ID, {'_', Pid}).
```

改写后的版本如下：

```
delete(Pid) ->
    case mnesia:dirty_index_read(key_to_pid, Pid, #key_to_pid.pid) of    %% ❶ 按 pid 查询表项
        [#key_to_pid{} = Record] ->
            mnesia:dirty_delete_object(Record);
        _ ->
            ok                                                           %% ❷ 查询出错也应返回 ok
    end.
```

在执行删除操作时可能会出现两种情况：要么键原本就不存在（可能已经被删掉了），要么键存在并被成功删除。无论是哪种情况，都应该返回ok。删除操作具有幂等性——也就是说同一

操作反复执行多次，结果相同。如果能找到与给定的键相对应的key_to_pid记录，将之删除，否则就直接返回ok❷。

请仔细看看我们是怎样仅凭pid来定位相应的表项的❶。这段代码展示了Mnesia索引的用法：操作索引时需要使用一些专用于索引的函数（上面用到的这个函数属于脏操作函数）。以index_read/3为例，它的第一个参数是表名，第二个参数是待检索的键（即pid），第三个参数则用于指定希望使用哪个索引来进行检索（一张表可以有多个索引）。指定索引时用的是索引字段的编号，该编号可以通过#recordname.fieldname语法获得。此外，还可以通过在建表时给定的索引名（即原子pid）来指定索引，但这样做会略微拖慢操作的执行速度。除去这些区别，该函数与普通的读取操作一般无二。

还不算太复杂，是吧？现在，缓存应用的存储系统已经完全由Mnesia接管，而且在整个改造过程中我们只修改了sc_store模块，该模块以外的代码一行都没有改。在后续的分布式学习中，你能否继续这种简洁明快的设计风格呢？让我们拭目以待吧。

9.3.2　让缓存识别出其他节点

下一步，我们需要让集群中的所有缓存实例都能识别出彼此。这实际上是在为跨实例数据同步修桥铺路，走完这一步才能实现所有实例间的数据共享。对于刚启动的缓存节点来说，第一要务就是尽快加入集群。在这一小节中，针对这一问题我们给出了一种简便的解决办法。节点加入集群的途径有很多，而我们提供的方案可谓既简单又务实。

在该方案下新节点将通过两个长期运行的空白Erlang节点来加入预先约定的集群。这两个节点不执行任何用户代码（因而几乎永远不会宕机）。你只需要启动它们，给它们分配恰当的节点名，并设置好用于集群认证的cookie就可以了：

```
erl -name contact1 -setcookie xxxxxxxx
erl -name contact2 -setcookie xxxxxxxx
```

新启动的缓存节点将按事先配置的节点名去ping这两个节点，如图9-10所示。图中的两个net_adm:ping/1调用只要能有一个成功，启动过程便能够得以继续；否则节点将无法加入集群，启动过程就此宣告失败并抛出崩溃转储文件。

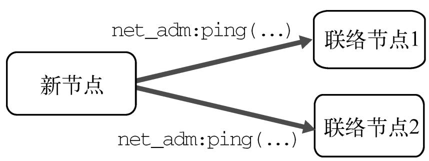


图9-10　新节点通过事先已知的、永久可用的联络节点自动加入集群：这个办法既简单又有效。两个联络节点最好分别运行于不同的物理机上

相关代码该放在哪儿就不用多说了吧？仔细想想应用的启动过程就一目了然了。应用启动时首先调用的函数是sc_app:start/2，6.4.2节中的sc_store:init()也是加在此处。由于在初

始化存储系统之前，节点之间必须连通，节点加入集群的代码自然也得放在这里了。

代码清单9-4列出了需要添加到sc_app中的代码。请注意在start/2中新增的那一行，这一行试图用ok去匹配ensure_contact()的调用结果。这实际上是一个断言：调用结果必须 是ok，否则就会触发一个badmatch异常。这是一条“不成功便成仁”的约定：要么如你所愿连接成功，要么start/2抛出异常，宣告应用启动失败。

代码清单9-4 在应用启动时加入集群（sc_app.erl）

```
start(_StartType, _StartArgs) ->
    ok = ensure_contact(),                         <-- 在 start() 中加入这一行
    sc_store:init(),
    case sc_sup:start_link() of
        {ok, Pid} ->
            {ok, Pid};
        Error ->
            Error
    end.

ensure_contact() ->
    DefaultNodes = ['contact1@localhost', 'contact2@localhost'],
    case get_env(simple_cache, contact_nodes, DefaultNodes) of   <-- ❶ 查询联络节点的配置信息
        [] ->
            {error, no_contact_nodes};
        ContactNodes ->
            ensure_contact(ContactNodes)
    end.
ensure_contact(ContactNodes) ->
    Answering = [N || N <- ContactNodes, net_adm:ping(N) =:= pong],   <-- ❷ ping 列表中的节点
    case Answering of
        [] ->
            {error, no_contact_nodes_reachable};
        _ ->
            DefaultTime = 6000,
            WaitTime = get_env(simple_cache, wait_time, DefaultTime),   <-- ❸ 查询等待超时的配置信息
            wait_for_nodes(length(Answering), WaitTime)
    end.

wait_for_nodes(MinNodes, WaitTime) ->
    Slices = 10,
    SliceTime = round(WaitTime/Slices),
    wait_for_nodes(MinNodes, SliceTime, Slices).   <-- ❹ 进入等待循环

wait_for_nodes(_MinNodes, _SliceTime, 0) ->
    ok;
wait_for_nodes(MinNodes, SliceTime, Iterations) ->
    case length(nodes()) > MinNodes of   <-- ❺ 检查是否已经连上了足够多的节点
        true ->
            ok;
        false ->
            timer:sleep(SliceTime),
            wait_for_nodes(MinNodes, SliceTime, Iterations - 1)
    end.
```

```
get_env(AppName, Key, Default) ->
    case application:get_env(AppName, Key) of          <-┐
        undefined   -> Default;                          ❻ 查询配置信息
        {ok, Value} -> Value
    end.
```

这堆代码看上去虽然多，但却一目了然。基本流程如下：

- 查询联络节点的配置信息（或采用硬编码的默认配置）；
- ping所有联络节点，收到应答后才能继续；
- 查询等待超时配置（或采用默认值），用于等待来自其他节点的连接；
- 等待来自其他节点的连接，一直等到连通节点数多于最初给出应答的节点数为止（或者一直等到不耐烦，然后假定自己是头一个加入集群的工作节点）。

和应用配置相关的问题我们留待下一章再详细讨论。至于现在，能有一个负责读取配置信息的函数就可以了❻。工具函数`get_env(AppName, Key, Default)`简单包装了`application:get_env(AppName, Key)`，在找不到对应配置时它会返回默认值。首次调用该函数的是`ensure_contact()`函数，目的是获取联络节点列表❶。此外，由于尚未加上配置信息，我们临时插入了两个硬编码的默认节点名。

拿到联络节点列表后，下一步就是调用`ensure_contact(Nodes)`，该函数将尝试用`net_adm:ping/1`联络所有节点❷。请注意这里用到了列表速构，既向所有节点发出了ping请求，又收集了所有应答节点的节点名，一举两得。如果所有节点都联络不上，就直接放弃。否则再检查一次配置，找出用于等待集群中所有节点达到全连通状态的超时时间❸（同样，为了应对配置缺失，此处也硬编码了一个默认值），然后默默地坐等其他节点的出现。

`wait_for_nodes/2`函数首先把等待时间切成了若干个时间片，然后进入等待循环❹。在每个时间片开始时，函数检查当前连通节点的总数是否大于最初给出应答的联络节点的总数❺。如果是，便假定集群中的所有节点都已连通。否则，就再等待一个时间片并重复上述检查。如果直到最长等待时间结束都没有新的节点出现，那也无妨，继续执行便可（在这种情况下，你可以假定自己是除联络节点以外第一个加入集群的节点）。

小心使用配置信息

用于读取配置文件的函数是不具备引用透明性的：传给函数的参数不再是函数返回值的唯一决定因素[①]。在函数式编程中，不要把这类逻辑埋藏到代码深处。把它们拎出来，布置到程序的最上层（放到初始化部分中），尽量打扮得显眼一些。（在上述场景中，只有`start/2`函数需要读取配置。）遵循这条实践原则，代码将更易于维护和重构。函数的引用透明性越强，代码的行为就越容易推断，调整代码时也就越容易判断影响范围。

有了这些代码，你便能够确保集群中的所有节点在监督树启动之前全部连通（应用的其余部分要在监督树启动之后才会依次展开）。下一步，我们要用资源探测来定位集群内的其他

① 还取决于配置文件的内容。——译者注

simple_cache实例。

9.3.3　用资源探测定位其他缓存实例

在这一节里，我们会把上一章中搭建的资源探测系统集成到simple_cache。资源探测是一个通用服务，无论是什么应用，只要它需要搜寻Erlang集群内的资源实例，都可以使用该服务。从这个角度考虑，你不会希望把资源探测系统硬塞在缓存应用之内。相反，我们应该把它组织成一个单独的应用，放在与simple_cache并列的目录结构中。

此处不再赘述应用的详细创建过程——整个过程你应该已经熟稔于心了，就留给你作为练习吧。和此前一样，你需要建立应用的目录结构，并编写相应的.app文件、_app模块，和_sup模块（用于启动8.3.3节中编写的资源探测服务器）。全部搞定之后，应该得到两个目录结构并列的应用，如下所示：

```
lib
  |- simple_cache
  |       |- src
  |       |- ebin
  |       |- ...
  |
  |- resource_discovery
          |- src
          |- ebin
          |- ...
```

此外，还需要给simple_cache增加两个依赖项，即resource_discovery和mnesia。代码清单9-5中列出了更新后的simple_cache.app，不仅加入了上述两个依赖项，还列出了第7章中的事件日志模块（请与6.3.2节中初始版本的.app文件进行对照）。

代码清单9-5　更新后的simple_cache.app文件

```
{application, simple_cache,
  [{description, "A simple caching system"},
   {vsn, "0.3.0"},
   {modules, [simple_cache,
              sc_app,
              sc_sup,
              sc_element_sup,
              sc_store,
              sc_element,
              sc_event,
              sc_event_logger]},
   {registered,[sc_sup]},
   {applications, [kernel, sasl, stdlib, mnesia, resource_discovery]},   <-- 加入了新的依赖项
   {mod, {sc_app,[]}}
]}.
```

框架工作奠定完毕，现在请仔细考虑一下我们所要实现的功能。一方面需要将本地的缓存实

例公布给他人使用，另一方面还需要定位集群中的其他缓存实例。回忆一下第8章中资源探测服务的工作原理，要实现这两点并不困难。要想进行发布，只需将待发布的本地资源放入资源探测服务器（用以表达“供给”什么资源），该资源应以simple_cache为类型，以节点名为资源引用（一个节点上最多只能有一个simple_cache实例，因而有节点名就足够了）。写成代码就是：

```
resource_discovery:add_local_resource(simple_cache, node())
```

要搜寻其他缓存实例，只需要告诉服务器“需求”类型为simple_cache的资源实例就可以了：

```
resource_discovery:add_target_resource_type(simple_cache)
```

最后一步就是向集群中的其余节点发起资源交换请求，接着耐心等待一段时间，直至资源信息交换完毕（这套资源探测系统具有较强的异步性，因而等待是必要的）：

```
resource_discovery:trade_resources(),
timer:sleep(?WAIT_FOR_RESOURCES),
```

很明显，这段代码也应该加到sc_app:start/2函数中，位置紧随ensure_contact()调用之后。详情参见代码清单9-6。

代码清单9-6　资源探测相关的修改（sc_app.erl）

```
-define(WAIT_FOR_RESOURCES, 2500).

start(_StartType, _StartArgs) ->
    ok = ensure_contact(),
    resource_discovery:add_local_resource(simple_cache, node()),
    resource_discovery:add_target_resource_type(simple_cache),
    resource_discovery:trade_resources(),
    timer:sleep(?WAIT_FOR_RESOURCES),
    sc_store:init(),
    case sc_sup:start_link() of
        {ok, Pid} ->
            {ok, Pid};
        Error ->
            Error
    end.
```

很简单吧？你已经甄别出了集群中的所有缓存实例，同时它们也都认出了你。现在只差最后一步了，还需要对Mnesia进行一些调整，以便与集群中其他的节点对接进而复制它们的数据。

9.3.4　动态复制Mnesia表

好啦，很快就要大功告成了。稍事休整，欣赏一下你的大作吧。细细回味一下：你的系统中几乎不存在任何静态配置——只有一些起引导作用的代码外加两个用于帮助新节点接入集群的联络节点——系统将会自动探测出所有simple_cache实例并复制它们的数据，整套系统高度动态，而且具备良好的容错性。

跨节点的冗余复制其实也没有多复杂，不过你还是得多了解一下Mnesia在分布式环境下的运作方式才行。和冗余复制相关的代码放在sc_store:init/0内，当启动过程进行到这一阶段时，

节点已经与集群连通，资源探测也已经完成（参见代码清单9-6）。这部分代码有一些难度，我们把它拆成了几个部分，然后逐一展开讨论。

将ETS替换成Mnesia之后的sc_store:init/0如下（参见9.3.1节）：

```
init() ->
    mnesia:start(),
    mnesia:create_table(key_to_pid,
                        [{index, [pid]},
                         {attributes, record_info(fields, key_to_pid)}]).
```

以此版本为基础，再加上资源探测：

```
init() ->
    mnesia:stop(),
    mnesia:delete_schema([node()]),
mnesia:start(),
{ok, CacheNodes} = resource_discovery:fetch_resources(simple_cache),
dynamic_db_init(lists:delete(node(), CacheNodes)).
```

该函数的第一要务是确保Mnesia正常启动，与此同时清理掉本地节点上现存的数据库模式。在数据面前，这段代码毫无顾忌，不假思索地将原数据库中的模式和表抹了个一干二净。不过没关系，毕竟这只是个缓存：所有的数据都是以ram_copies方式保存的——包括数据库模式在内。在删除数据库模式前，必须先停止Mnesia。（在Mnesia尚未启动时，调用mnesia:stop()是无害的。）

完成清理工作之后，init()会从资源探测系统中取出此前识别出的所有simple_cache实例。（前面曾经提过，我们是用节点名来区分这些实例的。）本地的缓存实例也包含在返回的缓存节点列表中，将它剔除，然后进入下一阶段：详情参见下列代码清单中的dynamic_db_init/1函数：

代码清单9-7 根据节点探测情况来初始化Mnesia

```
dynamic_db_init([]) ->
    mnesia:create_table(key_to_pid,
                        [{index, [pid]},
                         {attributes, record_info(fields, key_to_pid)}
                        ]);
dynamic_db_init(CacheNodes) ->
    add_extra_nodes(CacheNodes).
```

根据集群中是否存在其他缓存节点，该函数会采取不同的数据库初始化策略。函数的第一个子句用于处理整个集群中只有当前这一个缓存节点的情况，处理手法与9.3.1节相同。虽然清理完原有数据库模式之后还没有调用mnesia:create_schema/1，但实际上Mnesia已经在RAM中建立了全新的模式。一个完整的simple_cache实例就此启动完毕，后续加入集群的实例都将可以从它那里复制数据。

如果探测到集群中的其他simple_cache实例，就进入第二个子句。在这种情况下，我们转而从集群中的其他节点处拉取数据。相关逻辑由add_extra_nodes/1通过遍历所有远程节点来实现，如代码清单9-8所示。

必须有一个节点先行启动

有一点请千万注意，第一个启动的节点必须独自完成启动过程。如果两个simple_cache节点同时启动，就有可能会产生竞态条件，导致双方都认为对方是先启动的那一个。其后果就是初始数据库模式永远也建立不起来。为了避免这种情况，我们可以再加一些代码来进行同步。不过简洁起见，暂且先这样吧。

代码清单9-8　连接其他Mnesia节点并复制其上的数据

```
-define(WAIT_FOR_TABLES, 5000).

add_extra_nodes([Node|T]) ->
    case mnesia:change_config(extra_db_nodes, [Node]) of
        {ok, [Node]} ->
            mnesia:add_table_copy(schema, node(), ram_copies),     ❶ 用远程数据库的模式替换本地模式

            mnesia:add_table_copy(key_to_pid, node(), ram_copies),

            Tables = mnesia:system_info(tables),
            mnesia:wait_for_tables(Tables, ?WAIT_FOR_TABLES);
        _ ->
            add_extra_nodes(T)                                     ❷ 继续尝试其他节点
    end.
```

这个函数还真是干了不少事情，好在逻辑还算清晰。它首先调用了mnesia:change_config/2，让Mnesia再向数据库中添加一个节点。Mnesia的工作方式与Erlang节点类似：只要连上一个实例，就可以与所有实例连通，因此只要连上一个远程实例就行了。请注意，在上述情况下连接是由新的空白节点向存有数据的节点发起的，不过无所谓；Mnesia随后会更新本地的节点列表（远程节点上的也会更新），数据库节点至此添加完毕。（这种方式只能用于添加刚刚启动的、完全基于RAM存储，且数据库模式为空的节点。）

如果出于某些原因连接未能成功，就继续尝试列表中的其他节点❷。（如果所有节点都连不上，由于我们没有处理节点列表为空的情况，代码将会崩溃，进而导致启动失败。这又是一个放任崩溃的例子，在这种情况下你也的确是无能为力。）

如果连接成功，首先在本地节点上复制一份远程数据库的模式❶。该模式将会取代本地节点上临时的空白模式。接着再用同样的手法复制key_to_pid表。这才终于达成了你真正的目的——在缓存实例间共享key_to_pid表。最后，调用mnesia:system_info(tables)罗列出当前数据库中所有的表，再调用mnesia:wait_for_tables/2等待新表的内容同步完毕。（第二个参数表示超时，即便在这段时间内同步未能完成，启动过程也会继续。）

大功告成！看看吧：一套支持动态冗余复制的分布式缓存。放到shell中去跑跑看：启动两个节点，将它们连通，从一个节点插入数据，再从另一个上进行查询。在你构建的系统中，键与进程间的映射关系被存放在一张支持冗余复制的表中。有了这套系统，集群中任何缓存实例都可以在此存储映射关系，还可以查找与指定的键相关联的进程标识符，从而与对应的进程直接进行通信（无论进程位于哪个节点），进而读取或更新与键相关联的值。

运行该系统时，首先像先前一样在独立的终端窗口中启动一到两个联络节点（对于使用短节点名的集群请使用-sname选项）：

```
$ erl -sname contact1
```

事前请务必先把simple_cache和resource_discovery应用中的所有.erl文件都编译成.beam文件，并放入相应的ebin目录，譬如可以这样做：

```
erlc -o ./simple_cache/ebin ./simple_cache/src/*.erl
erlc -o ./resource_discovery/ebin ./resource_discovery/src/*.erl
```

然后按以下方式启动Erlang：

```
$ erl -sname mynode -pa ./simple_cache/ebin -pa ./resource_discovery/ebin
```

在启动simple_cache之前记得先启动其他几个它所依赖的应用（参见代码清单9-5中的.app文件）：

```
1> application:start(sasl).
ok
2> mnesia:start().
ok
3> application:start(resource_discovery).
ok
4> application:start(simple_cache).
ok
```

受资源探测功能的影响，simple_cache的启动过程会被耽搁几秒钟。如果simple_cache找不到联络节点，首先请确认是否所有节点都以-sname启动，然后检查sc_app:ensure_contact()中的默认节点名，如果localhost不管用，请将之修改成实际的主机名（改完之后记得重新编译）。

可以按同样的方式多开启几个节点（分别名为mynode1、mynode2……），每个节点分别占据一个独立的终端窗口，运行着相同的应用。然后试着在一个节点上写缓存，再从另一个节点上读取数据。看吧，行云流水。

9.4　小结

在这一章中，我们初步学习了Mnesia——一套极为灵活的分布式数据存储系统。在利用Mnesia改写缓存应用的存储后端的过程中，代码其余部分几乎没有受到任何影响。与此同时，我们还实现了一套简单的逻辑，确保缓存节点能够自动加入Erlang集群。最令人印象深刻的是，我们还集成了第8章中的资源探测系统，进而使集群中的所有缓存实例都具备了相互动态复制Mnesia表的能力。Erlware的家伙们一定很兴奋，会话存储的问题终于得到了解决。现在即便同一用户的请求被先后转发至不同的Web服务器，也不会出现糟糕的用户体验了。

在下一章中，此行终将迎来一个圆满的句号。我们会将这些代码整理成一份随时可发布到线上使用的发布镜像（release）。等迈出这一步，你便正式踏上Erlang专家之路了。

第 10 章

打包、服务和部署

10

本章概要

- 目标系统、应用和发布镜像
- 如何定义和启动发布镜像
- 发布镜像的打包和安装

时至今日，你已经学习了不少内容，包括OTP行为模式的运用、OTP应用的创建、日志和事件的处理、分布式应用的编写，还有Mnesia数据库的使用。下一个任务，就是把所有这些代码融为一体，为最终的部署做好准备。

OTP应用可以被视作一种功能单元，它们封装了各种功能，提供了一定的便利，但这种封装仅限于Erlang编程层面。要想打造出一整套完备而独立的软件服务——也就是那些在在网络中的一台或多台机器上运行，与其他系统或用户进行交互的东西——还需要把运行在同一Erlang运行时系统上的多个应用捆绑到一起。在OTP中，我们把这种更高层次的软件包称作*发布镜像*（release），在某台主机上安装发布镜像后，便形成了*目标系统*（target system）。

标准Erlang/OTP发行版中包含大量应用，其中不乏第5章中的那些图形化运行时工具。目标系统则不同，一般来说，它们只会包含自身提供服务所必需的那些应用。不过，每个目标系统至少都会包含stdlib和kernel这两个基础应用，出于日志方面的需求，往往还会包含SASL应用。

在详细讲解如何创建发布镜像之前，我们先回过头来仔细审视一下应用，看看在当前阶段这个概念还蕴含着哪些值得我们关注的要点。

10.1 从系统的角度看应用

OTP应用通常都是些具备自主运转能力的东西，它们由多个运行着的进程组成，运行时行为和生命周期也各不相同。与典型编程语言中的库相比较，它们更像是Web浏览器或Web服务器之类的独立应用软件。在启动OTP应用时，一般会同时启动多个长期运行的进程，其中某些进程甚至会随着整个系统一直运行下去。

Erlang的目标系统由多个运行中的应用组成。这些应用有着类似的结构和元数据，管理方式也一致。

10.1.1 结构

所有应用都具有相同的基本结构，如图10-1所示。`application`行为模式封装了应用的启停。启动之后，每个应用在运行时都会有一个根监督者，直接或间接地管理着应用中的所有其他进程（其中也包括子系统的监督者）。

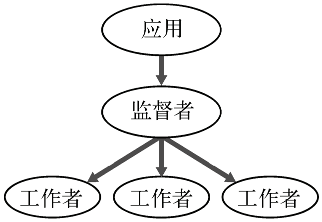


图10-1 应用在运行时的一般结构。所有OTP应用的启停方式和管理方式都是统一的。这大大简化了将应用打包成发布镜像的过程

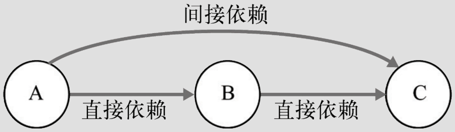
应用意味着一致性——它统一了行为模式的打包和管理方式。所有应用都有着标准的目录结构、定义清晰的入口，以及规范的监督结构。在将多个独立应用打包成发布镜像的过程中，这种一致性使得大部分步骤得以自动化。

10.1.2 元数据

大部分应用都是自主运转的，而不仅仅是静态的库。为了了解应用的启动方式、应用之间的依赖关系等知识，OTP需要掌握一些信息。这些信息就包含在.app文件中。在我们当前的项目中有两个应用：resource_discovery和simple_cache。它们又分别依赖于kernel、stdlib、sasl和mnesia等标准OTP应用。

依赖关系与传递性

应用之间往往会存在依赖关系。譬如simple_cache应用直接依赖于kernel和stdlib应用（实际上所有应用都直接依赖于这两个应用），以及mnesia、resource_discovery和sasl应用。推而广之，应用之间也可能存在间接依赖。如下图所示，假设应用A依赖于应用B，而B又依赖于应用C，鉴于依赖关系的传递性，我们说应用A间接依赖于应用C。



依赖关系的传递性。由于应用A直接依赖于应用B，而应用B又直接依赖于应用C，因此应用A间接依赖于应用C

simple_cache当前的.app文件看起来应该是这样的：

代码清单10-1　应用元数据：simple_cache.app

```
{application, simple_cache,
 [{description, "A simple caching system"},
  {vsn, "0.3.0"},
  {modules, [simple_cache,
             sc_app,
             sc_sup,
             sc_element_sup,
             sc_store,
             sc_element,
             sc_event,
             sc_event_logger]},
  {registered, [sc_sup]},
  {applications, [kernel, sasl, stdlib, mnesia, resource_discovery]},
  {mod, {sc_app, []}}
 ]}.
```

从应用管理的角度看，这些元数据都是不可或缺的。其中有一项元数据对发布镜像尤为重要，那就是vsn元组，用于指定应用的当前版本。

10.1.3　系统如何管理运行中的应用

sc_app等application行为模式的实现模块就是OTP应用的启动入口（参见.app文件）。这些模块提供了应用的启停接口；此外，还有一个与application行为模式对应的行为模式容器，叫作*应用控制器*（application controller），系统中当前运行着的所有应用都是由它来管理的。应用的start/2回调函数应该在应用启动完毕后返回新启动的根监督者进程的pid，容器将通过该pid来追踪应用。从这个角度来看，application行为模式与第7章中的gen_event行为模式很类似：二者都采用单个容器管理行为模式的多个实现模块（参见图7-1）。

1. 应用控制器

每个运行时系统中只有一个应用控制器，其注册名为application_controller。从下列示例中可以看出，该进程的标识符数值很小，可见在Erlang运行时系统的启动过程中，应用控制器很早就被启动起来了：

```
Eshell V5.7.4  (abort with ^G)
1> registered().
[kernel_sup,global_name_server,inet_db,init,file_server_2,
 code_server,erl_prim_loader,user_drv,standard_error,
 application_controller,error_logger,kernel_safe_sup,user,
 global_group,standard_error_sup,rex]
2> whereis(application_controller).
<0.6.0>
```

控制器还负责加载应用的.app文件，并且会检查当前应用所依赖的其他应用是否都已经启动。应用控制器会为运行中的每个应用派生一对应用主控进程，从而把自己与应用代码隔离开来（参见图5-3）。目前而言，你还不必关注这些额外的进程，不过仍然有必要了解它们是容器整体功能的一部分——尤其是在调试的时候。尽管application行为模式容器的内部结构比大多数行为模式都要复杂，但application模块提供的API却很简洁。

2. 应用的启动类型

在用`application:start(AppName)`启动应用时，应用的启动类型默认为`temporary`。也就是说，就算应用意外终止，运行时系统的其他部分也不会受到影响，只会生成一份崩溃报告。然而，如果通过调用`application:start(AppName, permanent)`启动，应用就会被认为是目标系统中不可或缺的组成部分：无论出于什么原因，只要应用终止，整个运行时系统也会随之关闭，只能重新启动。（还有一种启动类型是`transient`，但对于普通OTP应用来说，它和`permanent`是一样的。）系统的重启可以交由外部操作系统中的心跳进程来处理，详情请参见Erlang/OTP文档中的`heart`模块（它是kernel应用的一部分）。

在这一节中，我们从系统的角度强调了应用的一些要点，现在让我们来看看如何将应用组织成发布镜像，进而构筑独立的Erlang系统。

10.2 制作发布镜像

Erlang/OTP的功能封装是分层的。位于最底层的是模块，用于封装代码。模块进而会被组织成应用，用于对动态的行为模式和更为高级的功能进行封装。最终，应用又会被进一步组织成发布镜像。

10.2.1 发布镜像

若干应用，再加上一些元数据，便构成了发布镜像。其中的元数据用于描述如何以系统的方式启动和管理这些应用。同一发布镜像中的应用都在同一套Erlang运行时系统上运行，这就是目标系统。目标系统是经过裁剪的运行时系统，只能用于运行发布镜像中的应用。从这个角度讲，发布镜像也可以被当作一种定义服务的手段：将运行中的Erlang VM视作一个系统级服务，正如Web服务器将simple_cache视作Web服务器的一个黑箱服务一样。

发布镜像中不仅包含用于实现其主要功能的应用，还包含这些应用直接或间接依赖的其他应用。譬如，在发布镜像中肯定会有simple_cache和resource_discovery，二者分别依赖于若干其他应用，这些应用又进一步依赖于更多应用，所有这些应用全部都要塞到发布镜像里去。

除了指定应用的版本号，发布镜像自身也有一个版本号。例如，图10-2中的发布镜像simple_cache-0.1.4中包含应用simple_cache-0.3.0、resource_discovery-0.1.0、kernel-4.5.6和stdlib-6.0.5。版本号是发布镜像的重要属性之一。

总结一下就是：

- 发布镜像描述的是可运行的Erlang运行时系统；
- 每个发布镜像都有一个版本号；
- 发布镜像是由一系列带版本号的应用和用于描述系统管理策略的元数据共同构成的；
- 将发布镜像安装到主机上，就形成了目标系统。

在Erlang/OTP的整个生态系统中，发布镜像是一个非常重要却又经常被人误解的概念。下面，我们就来带你一起创建simple_cache的发布镜像，很快你就会发现它根本没有传说中的那么复杂。

10

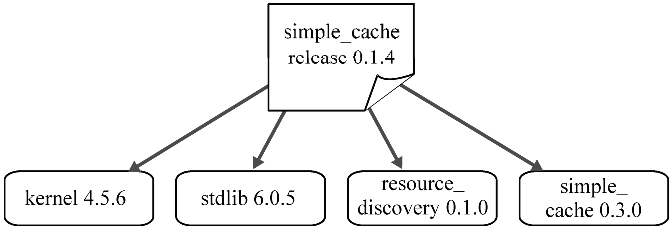


图10-2　发布镜像和版本控制。每个发布镜像都带有一个版本字符串，其中每个应用又各有一个版本号

10.2.2　准备发布代码

创建发布镜像的一般步骤如下：

(1) 确定需要包含哪些应用；

(2) 创建用于描述发布镜像内容的元数据（.rel）文件；

(3) 创建启动脚本；

(4) 创建系统配置文件（可选，但一般都会需要）；

(5) 将所有内容打包成单个文件。

我们将带你逐一完成这些步骤。在了解确切做法之后你便会发现整个过程一点儿都不难。首先，选出需要囊括到发布镜像中的应用。

目前，我们自行开发的应用一共有两个：simple_cache和resource_discovery。对于一套目标系统而言，要想将simple_cache当作服务来运行，这两个应用缺一不可（因为simple_cache用到了resource_discovery）。除此以外，被它们直接或间接依赖的应用也必须囊括在内，包括stdlib、kernel、sasl，还有mnesia。接下来，就该创建.rel文件了。

10.2.3　发布镜像的元数据文件

正如每个应用都带有一个用于存放应用元数据的.app文件，发布镜像里也有一个用于存放元数据的发布镜像文件，其扩展名为.rel。发布镜像元数据的主要内容是一张列表，其中包含该发布镜像中的所有应用。此外，还有一些内容需要指定，譬如，为了保证应用能在同一版本的Erlang运行时系统（ERTS）下编译和运行，必须指定ERTS的版本号。

在开发过程中，发布镜像文件放在哪里都无所谓。毕竟，只有在构建发布镜像时它才会发挥作用。也许你会把它放到与代码完全无关的地方。你也有可能会用到多个发布镜像文件（文件名不同），以便从同一份代码中创建出不同的发布镜像包。甚至，你还可以用别人发给你的代码来创建发布镜像。

如9.3.3节所述，当前的目录结构中（至少）应该含有两个应用的子目录：

```
lib
 |- simple_cache
 |    |-ebin
```

```
|    ...
|
|- resource_discovery
|    |-ebin
     ...
```

父目录名不一定要叫lib，但不管这个目录叫什么，在本章中，它就是存放.rel文件以及其他跟发布镜像相关的文件的地方。在运行后续的各个示例时，请将它设为当前目录。等你搞清楚了发布镜像的工作机制，便可以在实际的开发过程中视自己的工作流程需要来进行调整了。

代码清单10-2展示的是simple_cache.rel文件的内容，它将成为我们创建发布镜像的基础。首先请确保resource_discovery和simple_cache的版本与.app文件中相应的应用版本相同。根据所安装的Erlang版本的不同，你可能还需要调整erts、kernel、stdlib、sasl和mnesia的版本（现在暂时先别动它们）。

代码清单10-2　simple_cache.rel

```
{release,
 {"simple_cache", "0.1.0"},
 {erts, "5.7.2"},
 [{kernel, "2.13.2"},
  {stdlib, "1.16.2"},
  {sasl, "2.1.5.3"},
  {mnesia, "4.4.10"},
  {resource_discovery, "0.1.0"},
  {simple_cache, "0.3.0"}
 ]}.
```

和.app文件一样，.rel文件中也是一个由句点结尾的Erlang元组。这个元组由4个元素组成。其中第一个是原子`release`。紧随其后的是由发布镜像的名称（是字符串，不是原子）及版本构成的二元组。和应用的版本一样，发布镜像的版本字符串内容任意，但是考虑到用户的感受和对各种第三方工具的兼容性，最好还是遵循传统的版本字符串格式吧。

第三个元素用于指定ERTS的版本。这是一个二元组，由原子`erts`和所需要的Erlang运行时系统的版本字符串组成。请注意，ERTS的版本并非Erlang/OTP发行版的版本（譬如R13B03）。要查看ERTS的版本，请启动Erlang shell：

```
$ erl

Erlang R13B03 (erts-5.7.4) [smp:2:2] [rq:2] [async-threads:0]

Eshell V5.7.4  (abort with ^G)
1>
```

输出结果的第一行中，紧随文本Erlang R13B03之后的就是ERTS的版本，即5.7.4。查看版本的另一个办法是呼出BREAK菜单（参见2.1.4节）后进入选项`v`：

```
1>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
Erlang (BEAM) emulator version 5.7.4
Compiled on Tue Nov 24 11:12:28 2009
```

现在，请根据自己当前安装的系统版本对.rel文件中的erts项进行修改。

.rel元组中的第四个（同时也是最后一个）元素是一张列表，其中包含了发布镜像中的所有应用的名称和版本。在这里，我们不仅列出了主应用simple_cache，还列出了它的直接依赖和间接依赖：resource_discovery、mnesia、sasl，甚至还包括stdlib和kernel应用。在此必须完整地列出目标系统所需的所有应用，不仅包括你编写的应用，也包括它们直接和间接依赖的所有应用。要想确定其余应用的版本，最简单的办法就是参照下一节中make_script函数的执行结果。

.rel文件只是一个整体规范。运行时系统无法在启动时读取它——即便能够读取，它也无法提供足以让系统正常启动运行的信息。.rel文件并未指出ERTS可执行文件的位置，也没有说明到哪儿才能找到文件中列出的各个应用。对于真正的Erlang/OTP目标系统而言，这些信息应该在系统启动前准备就绪才行。接下来，我们将教你如何将这些信息与配置文件等其他必需品打包到一起，进而创建出可用于立刻启动目标系统的完整的发布镜像规范。

10.2.4　脚本与启动文件

.rel文件是我们迈出的第一步，距离真正可运行的Erlang/OTP目标系统还有一段距离。接下来，为了更加完善地描述系统的启动过程，还需要创建两个文件，它们就是.script和.boot文件。其中.script文件内包含一份完整的规范，所有应用的内容明细全部都罗列在内，包括应用的路径、需要加载的模块，以及其他各种必要信息。.boot文件则是.script文件的二进制形式，可供Erlang运行时系统在启动时直接读取。

要想创建这两个文件，必须先启动一个Erlang VM，.rel文件中提及的所有应用的路径都必须在VM启动前设置好。对于打算纳入发布镜像却又不在默认路径下的应用（也就是除Erlang/OTP自身以外的所有应用），可以用-pa命令行参数（参见4.3节）来将之加入代码路径。以下命令可以在启动shell的同时将simple_cache和resource_discovery应用的ebin目录加入代码路径。请在.rel文件所在的目录下执行该命令：

```
erl -pa ./simple_cache/ebin -pa ./resource_discovery/ebin
```

下一步就是调用systools模块（SASL应用的一部分）来生成真正的.boot和.script文件，方法如下：

```
1> systools:make_script("simple_cache", [local]).
ok
```

该命令执行完毕后，当前目录下会生成两个文件：simple_cache.script和simple_cache.boot。（如果命令报错称.rel文件中的版本号不对，请根据自己本地系统的情况更新.rel文件，然后再次执行该命令。）

感兴趣的话，可以打开.script文件瞧一瞧——里面的内容大致如下：

```
%% script generated at {date} {time}
{script,
    {"simple_cache", "0.1.0"},
    [{preLoaded, [...]},
     ...
```

该文件由数百行Erlang项式构成，完整地描述了系统的整个启动过程。你不一定非得搞明白这些内容，不过要是修改了.script文件，请记得调用`systools:script2boot(Release)`重新生成.boot文件。

local 参数主要用于测试

在调用 `make_script/2` 函数时，传入的 `local` 选项的作用在于将所有应用的绝对路径写入.script 和.boot 文件，也就是说，在用这份.boot 文件启动系统时，所有相关应用的位置必须与创建.boot 文件时各应用所在的位置相同。对于测试来说 `local` 选项很合适，但这种方式的可移植性不太好，不一定适合于生产环境。

传给`make_script/2`的`local`选项主要用于免除安装过程，以便简化针对系统启动过程的测试（我们很快就要开始了）。如果不采用`local`选项，生成的.script和.boot文件会认为所有应用都位于文件系统中的某个名为lib的目录下，该目录的具体位置由系统变量`$ROOT`指定。在需要将发布镜像安装至多台路径不尽相同的主机上时，这种方式尤为合适；不过就目前而言，采用`local`选项更加方便——我们的应用还在开发中，尚未达到能够在文件系统中的任意位置落地的阶段。

10.2.5 系统配置

simple_cache发布镜像就快创建完毕了。就当前这个发布镜像而言，最后一件需要操心事情就是配置文件。你应该还记得，在第9章中我们加入了用于读取配置的代码，但当时生效的实际上是代码中的默认值（参见代码清单9-4）。现在，该来创建发布镜像的配置文件了。

配置文件的标准文件名为sys.config。实际上只要扩展名是.config，文件名叫什么无所谓。跟.app文件和.rel文件一样，.config文件的内容也是一个由句点结尾的Erlang项式。simple_cache的sys.config文件内容如下所示：

代码清单10-3 sys.config配置文件

```
[
 %% write log files to sasl_dir
 {sasl,
  [
   {sasl_error_logger, {file, "/tmp/simple_cache.sasl_log"}}      <-- ❶ SASL 日志路径
  ]},
 {simple_cache,
  [
   %% Contact nodes for use in joining a cloud
   {contact_nodes, ['contact1@localhost', 'contact2@localhost']}  <-- ❷ 联络节点的节点名
  ]}
].
```

在每个.config文件中，项式的最外层都是一张二元组列表。其中的每个二元组都对应于一个应用，由应用的名称和一张表示应用选项的键值对列表构成。上面的sys.config文件指定了sasl应用的错误日志输出位置❶以及simple_cache应用中联络节点的节点名❷（请与9.3.2节中的代码进

行对比）。

现在，发布镜像中的各个组成部分都已经准备就绪，如图10-3所示。准备启动它吧。

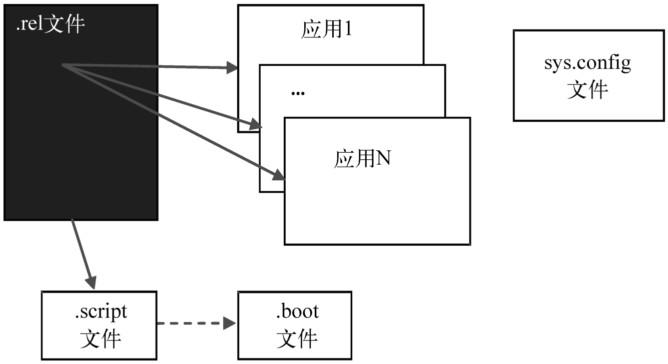


图10-3　发布镜像的各个组成部分。.rel文件指明了需要包含的应用，用于生成.script和.boot文件。sys.config文件是可选的，但通常都会用上

10.2.6　启动目标系统

现在，必要部件已经齐备，可以启动系统了。在着手启动之前，还需要指定两样东西：即启动系统时应该使用哪个.boot文件和哪个.config文件。在本例中，这两个文件都位于目标系统的根目录下。因此，应该使用下列命令启动系统（简便起见此处采用了`-sname`选项）：

```
erl -sname cache -boot ./simple_cache -config ./sys
```

不要忘了，还需要至少启动一个联络节点（参见9.3.2节）才行。联络节点的节点名务必要与代码清单9-4中硬编码的默认值或sys.config中配置的节点名保持一致。这些节点也要用`-sname`启动（不能用`-name`），否则它们将无法与缓存节点共同构成集群。

执行上述命令将会打开一个Erlang shell，这就表示系统启动成功了。SASL的输出被重定向至sys.config指定的文件之中，因而启动过程中的输出不再像以前那样琐碎——可以通过核实日志文件的内容来确认系统是否运作正常。在生产环境中启动目标系统时，通常你并不期望打开shell，而是期望系统以守护进程的形态在后台运行。（可以在其他节点上通过远程shell来登录在后台运行的系统，参见8.2.6节。）这也不难，只需在命令行中加上`-detached`选项即可。

```
erl -sname cache -boot ./simple_cache -config ./sys -detached
```

不过就目前而言，在前台运行的shell会话才是我们需要的东西，这样才能快速验证发布镜像的运行状况。要进行验证，可以使用Appmon（参见5.1节）。从运行目标系统的shell中启动Appmon：

```
1> appmon:start().
{ok,<0.72.0>}
```

这样就能看到Appmon的主窗口了，如图10-4所示。

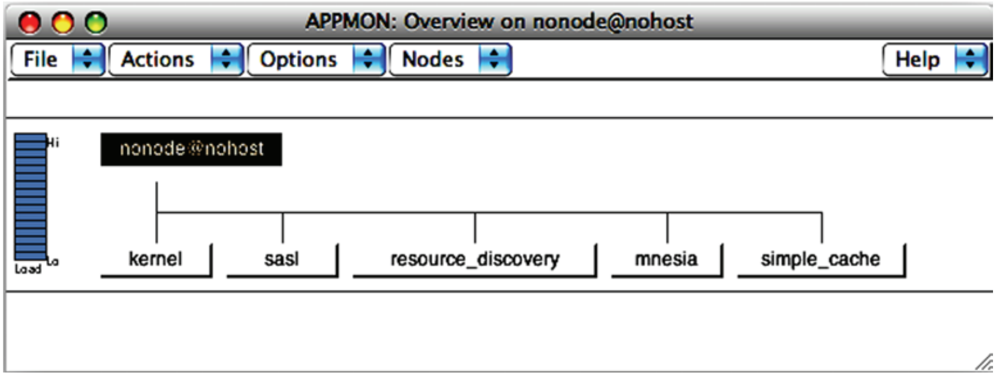


图10-4　Appmon的主窗口，展示在发布镜像中运行的应用

如你所见，在simple_cache.rel中作为依赖项列出的所有应用都启动起来了（除了身为库应用的stdlib）。大功告成，Erlang目标系统运行成功，太酷了！

目标系统还有一种启动方法，就是在命令行中加入`-mode embedded`选项。在默认交互模式下，Erlang会按需动态加载模块，你可能已经习以为常了。但对于嵌入式目标系统而言，这种模式却不一定适用，甚至它们有可能根本不支持在运行时加载其他代码。当Erlang系统以嵌入式模式启动时，它会在启动的同时根据启动脚本加载所有代码；此后，若要加载其他模块或是调用未曾加载过的模块，都会失败。详情请参见Erlang/OTP官方文档。

还可以试试`-detatched`选项。采用这种方式启动的系统将在后台运行，不会打开shell。不过，如果在联络节点的shell内调用`nodes()`，仍然可以在结果中看到分离的节点，可以针对该节点启动一个远程shell（参见8.2.6节）来对它执行各种操作，比如调用`init:stop()`便可关闭节点。

在下一节中，我们将把新鲜出炉的发布镜像打包成单个文件，从而简化安装和部署过程。

10.3　发布镜像打包

现在，发布镜像已经定义妥当，为了简化后续的安装、分发和部署过程，可能需要对它进行打包。OTP提供了若干实用功能来支持发布镜像的打包操作，尽管如此，往往还是免不了要做一些人工调整——一键打包什么的就别想了（至少暂时还不行）。出于这个原因，人们又开发了各种用于辅助打包和分发过程的工具。不过在这一章中，我们只关注如何利用OTP自身提供的功能来创建和安装发布镜像包。

10.3.1　创建发布镜像包

systools另有一个专门用于创建发布镜像包的功能，这就是`make_tar()`。熟悉tar的话就能猜到，`make_tar`的作用无非就是把发布镜像包中的所有文件合并成一个tarball。

什么是 tarball

tar 是 UNIX 系统提供的一个通用的文件归档工具。tar 和 zip 类似，但它不会对文件进行压缩，它只会把文件合并到一起形成单个称为 tarball 的文件，该文件更易于复制，也便于在其他位置解包。Tarball 文件的扩展名为.tar。

一般情况下，还会用 gzip 工具再把这个文件压缩一遍。这样的话，最终生成的文件的扩展名将是.tar.gz 或.tgz。在 Windows 下，可以用 7-Zip 等程序来创建、解压或查看 tar 文件。Erlang stdlib 应用中的 erl_tar 模块提供了对 tar 文件的支持。

和此前执行systools:make_script/2时一样，必须先根据正确的应用路径打开一个Erlang shell。去掉local选项后再执行一遍make_script/2，便可创建出更为实用且位置无关的启动脚本。然后，以发布镜像文件的文件名（不带扩展名）为参数调用make_tar/2：

```
$ erl -pa ./simple_cache/ebin -pa ./resource_discovery/ebin

Eshell V5.7.4  (abort with ^G)
1> systools:make_script("simple_cache", []).
ok
2> systools:make_tar("simple_cache", [{erts, code:root_dir()}]).
ok
```

其中的erts选项表示运行时系统（ERTS）也要一并打包在内，这样创建出的发布镜像可以在任何兼容主机上安装和启动。ERTS位于Erlang安装目录的根目录下（可通过code:root_dir()定位），在本例中，ERTS将被从该位置复制出来。如果不将ERTS打包在内，就必须在目标机器上单独安装ERTS（或者全套Erlang/OTP发行版），而且必须选用与发布镜像包的需求相匹配的版本。

附带 ERTS 会加深发布镜像包对 OS 的依赖性

将 ERTS 纳入发布镜像包，就意味着 ERTS 中的可执行文件也会被一并纳入，而这些可执行文件只能在与自身兼容的操作系统上运行。例如 32 位 Linux 上的可执行文件是无法在 64 位 Linux 系统上运行的。如果安装到不兼容的机器上，启动系统时很可能会看到“erlexec: no such file or directory”等古怪的错误信息。

现在检查一下当前目录，你会发现一个新的名为simple_cache.tar.gz的文件。这便是经过压缩的、包含发布镜像中所有文件的tarball文件。

10.3.2　发布镜像包的内容

一起来看看tarball文件中都有些什么，可以说OTP中最重要的内容都悉数涵盖在内了。在类UNIX系统上，可以用下列命令进行解压，解压后的文件位于名为tmp的新目录中：

```
$ mkdir tmp
$ cd tmp
$ tar -xzf ../simple_cache.tar.gz
```

如果手头没有tar程序，也可以在Erlang shell中进行解压：

```
2> erl_tar:extract("simple_cache.tar.gz", [{cwd, "tmp"}, compressed]).
```

无论采用哪种手段，最终你都将得到以下目录结构（根据所安装的Erlang/OTP的版本的不同，此处列出的应用版本号也可能会有所不同）：

```
tmp
|-- erts-5.7.4
|   `-- bin
|       |-- erl.src
|       ...
|-- lib
|   |-- kernel-2.13.2
|   |   ...
|   |-- mnesia-4.4.10
|   |   ...
|   |-- resource_discovery-0.1.0
|   |   |-- ebin
|   |   |   ...
|   |   `-- priv
|   |-- sasl-2.1.5.3
|   |   ...
|   |-- simple_cache-0.3.0
|   |   |-- ebin
|   |   |   ...
|   |   `-- priv
|   `-- stdlib-1.16.2
|       ...
`-- releases
    |-- 0.1.0
    |   |-- start.boot
    |   `-- sys.config
    `-- simple_cache.rel
```

发布镜像包中一定会有lib和releases两个目录。此外，由于在调用`make_tar()`时加上了`erts`选项，目录结构中还会存在一个名为erts-<version>的顶层目录，该目录下还有一个名为bin的子目录，其中包含运行时系统中的所有可执行文件。请特别留意其中的一个名为erl.src的文件，这是一份erl启动脚本的模版（用于类UNIX系统）。在安装发布镜像时，应该将erl.src复制到erl中并将该文件中的特殊字符串`%FINAL_ROOTDIR%`替换成目标系统上的真实路径。

发布镜像所需的所有应用都包含在lib目录下。（默认情况下，这里只存放应用的ebin和priv子目录。）请注意，应用目录的目录名是包含版本号的。这样做是为了便于安装同一应用（甚至整个发布镜像）的多个版本，进而实现系统的动态升级——同样也适用于升级失败后的回滚。

releases目录用于存放与发布镜像相关的信息。除.rel文件以外，你还会在该目录下找到一个以发布镜像版本号命名的子目录。启动文件（.boot文件）和sys.config文件都被存放在内，不过启动文件被重命名成了start.boot。采用这种目录结构，安装在同一根目录下的多个发布镜像将可以共享这些文件。

Erlang/OTP安装程序采用的正是这种目录结构：到自己的电脑上看看，Erlang/OTP系统安装目录下的结构如出一辙（不过还会多出一些文件和目录）。

10.3.3　定制发布镜像包

还差一点点，发布镜像的创建过程就介绍完了。你往往还会需要在发布镜像包中加些东西，而这些事情systools:make_tar()又无法代为完成。例如，通常我们都再加上一个顶层目录bin，用于存放系统的各种安装脚本和启动脚本。下面，我们就在tmp下创建这个bin目录，再配上几个工具脚本（以下假设你使用的类UNIX操作系统）。第一个脚本是安装时用的，用于在发布镜像安装完毕后设置根目录，不妨管它叫bin/install：

```
#!/bin/sh
ROOT=`pwd`
DIR=./erts-5.7.4/bin
sed s:%FINAL_ROOTDIR%:$ROOT: $DIR/erl.src > $DIR/erl
```

请根据自己的发布镜像适当调整ERTS的版本号。接下来，再添加一个名为bin/simple_cache的启动脚本。脚本内容与我们在10.2.6节中启动系统时用的命令行类似：

```
#!/bin/sh
./erts-5.7.4/bin/erl \
    -sname cache \
    -boot ./releases/0.1.0/start \
    -config ./releases/0.1.0/sys \
    -detached
```

同样，请按实际情况适当调整版本号。当然，别忘了用chmod a+x ./bin/*给脚本打上可执行标记。

最后还有一件事，根据所选安装方式的不同，可能还需要修改发布镜像中版本子目录的目录名。OTP中的工具通常会做如下假设：单个目标目录下只会装有一种类型的发布镜像，并且目标目录一般都以发布镜像的名称命名（譬如/usr/local/simple_cache）；这么一来，就没有必要在版本子目录的目录名中再次重复这个名字——这就是上述示例中releases目录下的目录名是0.1.0的原因。但是，如果你打算在同一目标系统中创建多个更细粒度的发布镜像包，这种模式就不合适了。这时，应该抛弃Version格式的版本目录名，转而采用ReleaseName-Version格式的目录名（譬如，将releases/0.1.0改成releases/simple_cache-0.1.0）。这样一来，各个发布镜像中的应用便可以独立安装升级，而不必担心目录会重名了。

启动脚本中的路径也要同步修改。同时，请把.rel文件也移动或复制到版本的子目录中去，以免今后在该目录下解压新版本的发布镜像时覆盖原始文件。对于这种包，就没必要把ERTS也一并纳入了，只要再创建一个仅含ERTS的独立安装包就可以了。

等调整好发布镜像包中的文件，只需调用erl_tar模块再次生成tarball文件即可，如下所示（假设当前目录仍然是tmp目录）：

```
$ cd tmp
$ erl

Eshell V5.7.4  (abort with ^G)
1> erl_tar:create("simple_cache-0.3.0.tar.gz", ["erts-5.7.4", "lib",
➥"releases", "bin"], [compressed]).
```

这个命令执行完毕后，tmp目录下将会多出一个名为simple_cache-0.3.0.tar.gz的压缩包，你亲

手打造的发布镜像便栖身于此。（当然，乐意的话你也可以用tar命令来完成打包。）全功能的发布镜像包就此新鲜出炉，赶紧找一台主机把它安装妥当，然后快快启动它吧。

10.4　安装发布镜像

OTP通过SASL的release_handler模块为发布镜像的解包、安装和升级提供支持。这个话题相当复杂，我们并不打算在此做深入讨论。在实际应用中，这个模块的使用频率也并不高。这套方法无法应对安装在同一目标目录下的多套发布镜像，我们在上一节中为此所做的改动与release_handler的工作方式不兼容。在此，我们打算另行介绍一种解包和安装发布镜像的方法，这种方法更简单也更健壮。

由于发布镜像包已经将ERTS打包在内，随便找台兼容的主机，将它解压到任意任一目录下即可——既可以是空目录，也可以是先前安装的目标系统所在的目录（用于升级发布镜像），无须事先在机器上安装Erlang/OTP。下一步是解包。在类UNIX系统上，可以用tar；此外，无论用的是什么操作系统，都可以像之前那样调用erl_tar：

```
$ mkdir target
$ erl
Eshell V5.7.4 (abort with ^G)
1> erl_tar:extract("simple_cache-0.3.0.tar.gz", [{cwd, "target"},compressed]).
```

接着，用cd进入目标目录并执行先前创建的bin/install脚本设置好根路径。搞定之后，就可以用bin/simple_cache脚本启动系统了（记得至少先启动一个联络节点）：

```
$ cd target
$ ./bin/install
$ ./bin/simple_cache
```

请注意，系统是单独启动的，而且这套运行时系统仅包含必要的应用。所以Appmon用不了，不过可以通过联络节点来启动WebTool版的Appmon（参见5.1.2节），然后经由浏览器来观察系统。

自动化打包安装工具

本章所描述的过程中，有几个步骤免不了需要人工参与。为了解决这个问题，Erlang 社区内涌现出了一批自动化工具，基于这些工具又衍生出一批自动化的打包安装方法。本书的作者同时也是其中两套工具的负责人，这两套工具都可以从 http://erlware.org 获取。它们算不上什么标准方案，当然也不是完成这类任务的唯一工具。虽然难免有王婆卖瓜自卖自夸的嫌疑，但我们还是认为有必要在此提上一句。无论是用我们的自动化工具还是用别人的，都可以大大减轻负担——并且还可以有效规避人为误操作的风险。

10.5　小结

OTP发布打包的基本情况就介绍到这里。无论是运用现有的工具还是根据OTP的规范去打造

自己的工具，这些内容都应该够用了。这是一个里程碑——要知道，长久以来发布镜像都被视作是只有Erlang先知才有能力把控的暗黑魔法。

至此，你大概会觉得自己已经取得真经，大功告成了吧？模块、应用，还有发布镜像，全都学完了。还有什么别的吗？有。目标系统的作用往往需要在运行时通过与外界进行交互才能得以体现，其中不乏与同一台机器上的其他软件之间的通信。为了操控硬件，或是和Eclipse等Java GUI进行交互，这些软件可能会用C或Java来开发。这类交互便是本书第三部分的主题。

Part 3 第三部分

集成与完善

本书第三部分介绍 Erlang/OTP 应用如何与外界集成。不同的问题需要不同的工具来解决，为了适应异构生产环境下的开发工作，还有一些知识需要掌握。在这一部分，我们还会讨论性能问题，从而最大限度地发挥出 Erlang/OTP 程序的威力。

第 11 章

为缓存添加HTTP接口

本章概要

- ❑ 用Erlang开发高效的TCP服务器
- ❑ 为缓存应用添加一套基于简易文本协议的TCP接口
- ❑ 创建自定义OTP行为模式
- ❑ 开发一个基本的Web服务器
- ❑ 为缓存应用添加一套REST式HTTP接口

在先前章节中，我们实现了一套颇有亮点的缓存应用。Erlware团队计划在接下来的几个月中继续对它加以完善。他们意识到Simple Cache应用同样适用于Erlware之外的项目。然而该应用目前只有Erlang接口，因而也只能用于用Erlang开发的应用。这成了进一步推广Simple Cache的绊脚石。

现如今，生产环境中的服务往往由多种语言写成。在将Simple Cache开源之前，最好能让其他语言实现的系统也能享受到它带来的便利。按照这个思路，最自然不过的办法就是实现一套或若干套基于常见网络协议的接口。在这一章中，我们先为你安排了一个热身练习，实现一套基于TCP的文本协议；然后，你将自行开发一套简化的Web服务器，再经由该服务器为Simple Cache提供一套REST式接口！在这一过程中，你将习得从高级监督策略到自定义行为模式，乃至OTP内建HTTP协议解析函数的使用等一系列实用编程技巧。

在本章的第一部分中，你将学习如何编写高效的并发TCP服务器应用，并为缓存应用实现一套简单的文本协议接口。第二部分首先是一段针对HTTP协议和REST概念的粗浅介绍，然后开始学习自定义行为模式的创建，并实现一套基本的Web服务器，进而以此为基础为缓存应用搭建一套REST式接口。

本章就此拉开帷幕，我们先来简短讨论一下基于文本的通信协议以及如何在Erlang中高效运用TCP套接字。

11.1 实现 TCP 服务器

纯文本是一种通用载体。以纯文本为基础的协议不仅实现简单，而且易于使用和调试。为了

给缓存应用打造一套良好的TCP/IP接口，我们不妨先来实现一套基本的文本协议，这套协议应当尽量地简单和直观。你会发现这套协议的服务器与你在第3章中编写的RPC服务器颇为类似，但二者之间存在一个重要区别：此前的RPC服务器只能处理单个TCP连接。而在这一章中，你将要创建的是一套具备工业强度的服务器，这类服务器的创建是每个合格的Erlang程序员都应当掌握的必备技能。仰仗于Erlang内建的并发支持能力，这套服务器将可以轻松应对大量并发TCP连接。

11.1.1 高效TCP服务器的设计模式

对于需要并发处理多个请求的服务器，有一种有效的设计模式，那就是将服务器实现成由简易一对一（simple-one-for-one）监督者管理的`gen_server`。在6.3.4节中我们曾经做过介绍，简易一对一监督者的子进程都是动态创建的，而且这些子进程还必须同属一种类型。在本例中，服务器会在启动之初创建一个gen_server子进程，它将扮演TCP连接处理器的角色——这个进程一启动便阻塞执行`gen_tcp:accept/1`，开始监听新连接。建立新连接之后，这个`gen_server`会让监督者另行创建一个新的处理器进程，自己则转而为当前连接服务。新创建的进程实际上就是先前那个`gen_server`进程的克隆，监听新连接的任务将由它担负。

在这个模式下，接受新连接的时候`accept`和后续的连接处理之间几乎没有任何延迟；从上一个连接接受完毕到准备好接受下一个连接，期间的延迟也得以最小化。这个思路与其他编程语言中套接字的典型用法大相径庭，但却是最具Erlang特色的做法。图11-1便是这个设计中的通信流程及控制流程。

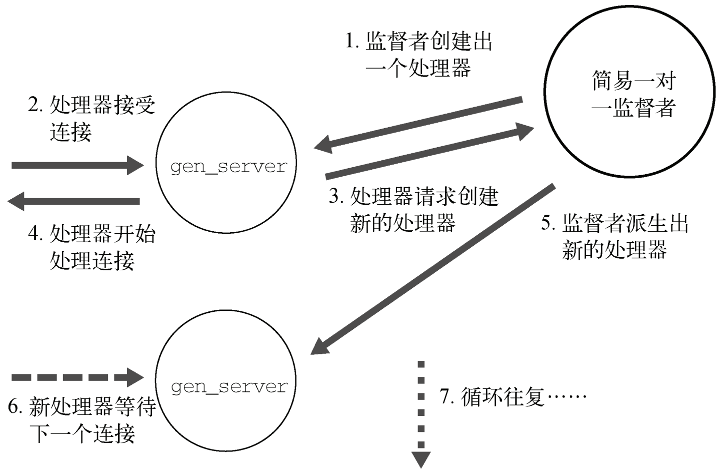


图11-1 用简易一对一监督者实现高并发TCP服务器。监督者会派生出多个子进程，每个子进程各自负责一个新TCP连接，全程掌管从`accept`到后续客户端通信的全部处理

看起来挺复杂，但很快你便会发现，用不了多少代码便能实现基本框架。（在Erlang中这是常有的事儿。）你应该想得到，要想搭建新的TCP服务器，还得先创建一个OTP应用。

11.1.2　搭建tcp_interface应用的骨架

Simple Cache可以有多套外部接口，这套TCP文本接口不过是其中的一个而已。为此，我们将它实现为一套单独的OTP应用。要是打算再添置一套接口，譬如UDP文本接口或是本章后续的HTTP接口，也请遵循这一模式，以免对现有代码产生干扰。在制作发布镜像（参见第10章）时，根据应用场景的不同，可以选择包含一套或若干套接口，也可以一套都不要。

现如今，想必你已经很熟悉OTP应用的创建步骤了（如有需要，请参阅第4章）。和其他应用一样，tcp_interface应用中有一个.app文件，一个应用行为模式实现模块ti_app，以及一个顶层监督者模块ti_sup。

过度简化的监督结构

出于简化代码的目的，此处我们采用上一节中提到的简易一对一监督者作为顶层监督者。不过在实际应用中，在这层之上一般还需要一层监督结构，正如第 9 章中位于 sc_element_sup 之上的 sc_sup。11.2 节中的 HTTP 接口将会采用更为稳固的监督结构。

除了上述内容，要实现图11-1中基于gen_server的处理器进程，还需要一个模块，这就是ti_server。这些文件就位之后，应用的目录结构应当如下所示（平行于simple_cache和resource_discovery应用所在的目录）：

```
tcp_interface
    |-- ebin
    |   `-- tcp_interface.app
    `-- src
        |-- ti_app.erl
        |-- ti_sup.erl
        `-- ti_server.erl
```

在本章接下来的内容中，我们将带你逐一实现这些模块的功能，事无巨细，无一遗漏。这一设计背后的简单、优雅、高效都将得到淋漓尽致的体现。在此期间我们将用到一些技巧，某些地方还会有些难度，不过等到本章结束时它们就都不成问题了。

11.1.3　填充TCP服务器的实现逻辑

我们将在11.2.1节中解释，要想接受TCP连接，必须先创建监听套接字。而且，持有监听套接字的进程在TCP服务器的整个生命周期内都必须活着；该进程一旦终止，监听套接字也将自动随之关闭。这样的话，除非再次创建新的监听套接字（譬如重启应用），否则服务器就无法接受新的连接了。

1. ti_app模块

那么在哪里打开监听套接字才好呢？根据OTP的设计原则，监督模块中的代码应当尽量精简；既然如此，不妨放在负责应用启动逻辑的ti_app模块中。打开的监听套接字可以经由此处传递给简易一对一监督者ti_sup，进而再扩散至新创建出来的各个处理器进程。

处理完监督者的初始化之后，ti_app模块还必须按图11-1的指示创建出首个连接处理器。代

码清单11-1便是`ti_app:start/2`的实现。

代码清单11-1 ti_app模块

```
-module(ti_app).

-behaviour(application).

-export([start/2, stop/1]).

-define(DEFAULT_PORT, 1155).

start(_StartType, _StartArgs) ->
    Port = case application:get_env(tcp_interface, port) of    ❶ 获取端口号
               {ok, P} -> P;
               undefined -> ?DEFAULT_PORT
           end,
    {ok, LSock} = gen_tcp:listen(Port, [{active, true}]),    ❷ 创建监听套接字
    case ti_sup:start_link(LSock) of
        {ok, Pid} ->
            ti_sup:start_child(),    ❸ 派生第一个处理器
            {ok, Pid};
        Other ->
            {error, Other}
    end.

stop(_State) ->
    ok.
```

模块首先在应用配置中查找服务器的监听端口号❶（参见10.2.5节）。如果没有指定端口号，则采用默认值1155。紧接着便是监听套接字的创建❷。然后，模块将以监听套接字为输入参数启动监督者。等到监督者运行起来之后，再调用`ti_sup:start_child()`❸创建出第一个ti_server进程，该进程将默默等待第一个连接的到来（图11-1）。

2. ti_sup模块

ti_sup模块与第6章中的sc_sup模块颇为类似，都是简易一对一监督者。二者之间的主要区别在于函数的参数个数：sc_sup（后来更名为sc_element_sup）中的`start_link`函数没有参数，`start_child`则有两个参数。本例中的`start_link`以监听套接字为参数，启动新子进程时则无须任何参数。ti_sup.erl的完整源码如代码清单11-2所示。

代码清单11-2 ti_sup模块

```
-module(ti_sup).

-behaviour(supervisor).

%% API
-export([start_link/1, start_child/0]).

%% Supervisor callbacks
-export([init/1]).

-define(SERVER, ?MODULE).

start_link(LSock) ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, [LSock]).
```

```
start_child() ->
    supervisor:start_child(?SERVER, []).

init([LSock]) ->                                                    ❶ 将套接字传入
    Server = {ti_server, {ti_server, start_link, [LSock]},            监督者的init回调
              temporary, brutal_kill, worker, [ti_server]},          ❷ 子进程规范中
    Children = [Server],                                               用到了套接字
    RestartStrategy = {simple_one_for_one, 0, 1},
    {ok, {RestartStrategy, Children}}.
```

start_link/1函数以监听套接字为参数，并将之传递给supervisor:start_link/3，进而最终传递至该模块自身的init/1回调❶。本例中的start_child/0函数很简单，无非就是让监督者按照init/1中的子进程规范另行创建一个子进程。传入init/1回调的监听套接字被写进了子进程规范❷，这样它便会成为所有新子进程的启动参数。这个简短的模块实际上就是一个既精巧又符合OTP规范的进程工厂，专门用于创建负责处理外来连接的ti_server进程。

3.ti_server模块

ti_server模块就是连接处理器，它负责从监听套接字处接受新连接，并给它们分配专用的套接字，从而与客户端建立TCP通信。如图11-1所示，在设计该模块时，我们采取的策略是让简易一对一监督者持有监听套接字，并将之传递给它创建出来的所有处理器进程。监听新连接的任务总是交由最新创建的处理器负责。一有连接接入，它就会让监督者再启动一个新的处理器并将监听任务移交过去，然后便转而着手处理自己刚刚接受的新连接。一旦走至这一步，处理器就再也不会恢复到监听状态了；从此往后，它将一直负责这一连接上的会话处理，并最终随会话的结束而终止。ti_server的初始版本如代码清单11-3所示。

代码清单11-3　基本的ti_server模块

```
-module(ti_server).

-behaviour(gen_server).

-export([start_link/1]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {lsock}).

start_link(LSock) ->
    gen_server:start_link(?MODULE, [LSock], []).

init([LSock]) ->
    {ok, #state{lsock = LSock}, 0}.                    ❶ 将超时置零

handle_call(Msg, _From, State) ->
    {reply, {ok, Msg}, State}.

handle_cast(stop, State) ->
    {stop, normal, State}.

handle_info({tcp, Socket, RawData}, State) ->
    NewState = handle_data(Socket, RawData, State),
    {noreply, NewState};                                   ❷ 处理收到的数据
```

```
handle_info({tcp_closed, _Socket}, State) ->
    {stop, normal, State};
handle_info(timeout, #state{lsock = LSock} = State) ->        ❸ 超时后跳至此处
    {ok, _Sock} = gen_tcp:accept(LSock),
    ti_sup:start_child(),
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

%% Internal functions
handle_data(Socket, RawData, State) ->             ❹ （暂时）将数据从套接字
    gen_tcp:send(Socket, RawData),                    原样返回
    State.
```

start_link/1函数就是监督者启动处理器进程并向新进程传入监听套接字的地方。随后，监听套接字又经由gen_server:start_link/3传至gen_server的init/1回调❶。init/1回调负责把套接字存入服务器状态记录，在返回的同时它还会将超时置零，以免阻塞init/1的调用者，我们曾经在第3章中用过这个手法（参见代码清单3-4和代码清单3-5）。将超时置零可以令新创建的gen_server进程的执行流程立刻落入handle_info/2的timeout子句❸。

至此，处理器进程终于摆脱了ti_server:start_link/1的调用进程（即ti_sup监督者进程），与早先启动又尚未终止的其他进程并行运行了起来。紧接着，这个处理器进程会立即在监听套接字上调用gen_tcp:accept/1，阻塞等待下一个TCP连接的到来。（请务必确保此时没有别的进程在等待该处理器进程，否则它们也会因为这个阻塞调用而被挂起。）

等到accept()返回时（返回时机视服务器的负载高低而定，有可能很快，也有可能由于接口基本没人用而要等上好几个月），首先便是调用ti_sup:start_child()让监督者再启动一个处理器。新的处理器进程本身是当前进程的一个克隆，它会立刻接过监听新连接的接力棒，同时，当前处理器进程转而继续处理刚刚接受的连接。

由于我们在打开监听套接字时采用的是主动模式（代码清单11-1），而且accept()返回的连接专用套接字会继承该属性，因此经由这些专用套接字传入的所有外来数据首先都会被转换成一条{tcp, Socket, RawData}格式的消息，然后再被自动发送至对应的处理器进程❷，个中细节我们曾经在第3章中做过介绍。当前版本的ti_server只会将外来数据经由TCP连接简单回传给客户端❹。最后，还需要处理套接字上的tcp_closed消息，从而确保ti_server进程能够随套接字的关闭而终止。

细节还真是多啊，不过物有所值，现在你已经拥有了一套通用的TCP服务器框架，只需加以调配便可适用于多种不同场景。以Simple Cache接口为例，只需对服务器的协议处理部分进行扩展便可。我们将在下一节讨论这些内容。

11.1.4 简单文本协议

本章的目的在于为Simple Cache应用提供更为丰富的对外接口。这样一来，无论采用的是何

种编程语言（甚至无论身处网络中的哪台机器），任何人都可以通过TCP与Simple Cache应用进行交互。截至目前为止，你已经有了一套尚不具备任何实际功用的TCP服务器框架。现在，我们将以这套框架为基础来实现一套简单的文本协议。

我们曾在第6章中做过介绍，simple_cache的API模块有3个导出函数：insert/2、lookup/1和delete/1（代码清单6-7）。它们便是TCP接口所要提供的功能。

外界在调用Simple Cache时，采用的协议可以表述为以下语法，其中|表示选择，Term表示Erlang项式常量：

```
Call -> Function ArgList

Function -> "insert" | "lookup" | "delete"

ArgList -> "[" "]" | "[" Terms "]"

Terms -> Term | Term "," Terms
```

举个例子:

```
insert[eric,{"Eric","Merritt"}]

lookup[eric]
```

另一方面，与每个请求相对应的应答的格式如下：

```
Result -> "OK:" Term ".\n" | "ERROR:" Term ".\n"
```

也就是一个以“OK:”或“ERROR:”为前缀、以句点加换行符为结尾的项式。例如，如果发送请求"lookup[eric]"，可能会得到以下应答：

```
OK:{"Eric","Merritt"}.
```

而当消息内容为"@*#$^!%"时，响应将会是：

```
ERROR:bad_request
```

insert操作应该有两个参数，分别是键和对应的值。lookup和delete操作则只需要键作为唯一参数。请注意键和值都可以是任意Erlang项式。

这套协议使用方便，解析简单。经过调配，也可以很容易地与其他类似的服务器配合使用。这是一套简单的请求/响应式协议，遵循协议规范的客户端每次只会发送一个请求，发完之后便坐等应答。这种模式可以有效控制每个连接处理器上的请求速率，从而将流量维持在可控范围以内。

客户端和服务器之间的消息格式已经定义完毕，可以开始实现请求的解析和处理逻辑了。

11.1.5　文本接口实现

实现这套简单文本接口，需要修改ti_server模块中的handle_data/3函数（参见代码清单11-3）。这份实现中的TCP套接字是以主动模式创建的，经由套接字接收的文本全都会被转换为消息，并自动投递给持有套接字的进程——也就是接受连接的那个处理器进程。这些消息将被传递给负责协议实现的gen_server:handle_info/2回调函数。为了令代码更规整，我们将消息处

理的部分全部挪入了内部函数handle_data/3，此前这个函数只是将收到的数据通过TCP套接字悉数奉还给客户端，不做任何处理。

新的代码需要完成以下步骤：

(1) 解析接收到的文本行；

(2) 将文本行转译为协议中的3个函数之一；

(3) 执行请求中的操作；

(4) 向客户端输出结果。

上述步骤可由以下代码完成。这段代码与第3章中的对应代码（代码清单3-6）很类似。

代码清单11-4 在ti_server中实现简单的基于文本的协议

```
handle_data(Socket, RawData, State) ->
    try
        {Function, RawArgList} =
            lists:splitwith(fun (C) -> C =/= $[ end, RawData),
        {ok, Toks, _Line} = erl_scan:string(RawArgList ++ ".", 1),
        {ok, Args} = erl_parse:parse_term(Toks),
        Result = apply(simple_cache, list_to_atom(Function), Args),
        gen_tcp:send(Socket, io_lib:fwrite("OK:~p.~n", [Result]))
    catch
        _Class:Err ->
            gen_tcp:send(Socket, io_lib:fwrite("ERROR:~p.~n", [Err]))
    end,
    State.
```

实际开发中应该将该函数进一步拆分成若干个辅助函数。但现在为了便于讲解，不妨暂且将它们揉为一体。（请注意，在实现这套简单协议时我们无须对服务器状态做任何修改，因此在函数的末尾处只需原封不动地返回State变量。）

第一步，以字符[为边界切分接收到的字符串。如果输入中的确包含[，则将之视作RawArgList部分的第一个字符；否则，RawArgList为空（这将导致后续解析的失败）。变量Function中应该是函数名——也就是字符[左侧的全部内容。

（根据上一节中定义的协议来看）字符串中RawArgList的那半段看起来和普通Erlang列表一般无二。这意味着可以直接将它传给Erlang的词法分析器erl_scan（需要预先补上一个句点），进而生成一张语元列表。将语元列表传给Erlang的解析器erl_parse，便可以解析出一张实际参数列表，这张列表自身也是一个Erlang项式。接下来就简单了，用内建函数（BIF）apply/3动态调用simple_cache模块中的具名函数即可。最后，通过TCP套接字回传结果，以便客户端读取。

整个过程中很多环节都有可能出错，我们采用try/catch表达式来对付这些错误，并把错误信息打印到TCP套接字中去。例如，当词法分析或语法解析失败时，运行时环境会因函数返回结果与{ok,...}不匹配而抛出异常；如果无法正常调用缓存应用中的函数（比如函数名不正确或根本就是个空字符串），又会抛出别的异常。

以上便是整个应用的完整实现——Simple Cache的TCP接口终于大功告成了！也许你会有点儿失望，心想："这就完了，就这么一段小程序！"可别忘了，你所实现的可是一套足以轻松抵御

数万并发连接的框架，而且你还可以随意对它进行扩展，构建自己的工业级强度的TCP服务器。

现在，不妨启动一套simple_cache系统来试试这套接口，步骤和上一章中一样，在shell中调用`application:start(tcp_interface)`即可，不过别忘了在路径中加上tcp_interface/ebin。（另外，在启动应用之前请务必先准备好ebin/tcp_interface.app文件。）调用`appmon:start()`确认所有应用都已经启动。等缓存和TCP接口都跑起来之后，可以按照第3章介绍的办法，用Telnet连入系统，然后输入11.1.4节中定义的命令。

还可以尝试一些会让第3章中的服务器一筹莫展的举动：利用两个或更多Telnet会话发起多个并发连接（比如从一个会话上写入数据再从另一个会话上读出来）。你给缓存应用加过日志，因此在通过TCP读写数据时应该能从Erlang终端上看到一些状态消息；但要是想观察TCP服务器进程内部的动静，那就必须给ti_server加日志了。最后，如果想让tcp_interface应用作为系统的一部分与系统一同启动，还可以把它写入发布镜像规范（参见第10章）。这个任务就留给你了。

这段小小的热身运动就此告一段落，我们该进入本章的第二部分了。这一部分中的目标更为雄心勃勃：我们将要实现一套REST式 HTTP接口，这套接口足以承载出入Simple Cache的任何载荷，并令缓存应用得以与任意REST式服务基础架构相整合。

11.2　打造一套全新的 Web 接口

自打本章一开始的时候我们就说过，在给缓存应用添加HTTP接口时，我们不但要以HTTP为基础来设计一套协议，还要搭建一套用于承载这套接口的真正的HTTP服务器。你可能会觉得这根本就是重度NIH综合征[①]的症状嘛！但这样做是有原因的。我们的目的有两个：首先，是要给出一个示例，它将为你展现一套完整而健壮的实用TCP服务器，顺便还会再讲解一些HTTP服务器和REST的相关知识（很快你便会发现，用Erlang来搭建Web服务器简直是小菜一碟儿）；其次，我们还要教你如何编写自定义的OTP行为模式——这里说的是将要用于实现HTTP接口的`gen_web_server`行为模式。不过在那之前，我们还是先对HTTP自身做一个了解吧。

11.2.1　HTTP简介

这本书不是讲HTTP的，所以我们不会浪费精力去讲解协议细节，更不打算介绍如何实现完整的HTTP服务器。在这一节中，我们会从HTTP协议中挑选若干最为关键的内容进行介绍。在用Erlang给Simple Cache实现高效的REST式接口时，有这些知识就够用了。

为了让你亲身体验一下HTTP，我们将会用上两个UNIX工具。第一个工具是用于观察TCP流量的nc（netcat）。利用它可以自行建立监听套接字，并对发往该套接字的所有数据进行观测。第二个工具是curl，这是一个命令行HTTP客户端，可以利用它向HTTP服务器发送任意请求。有了

① NIH综合征（Not Invented Here Syndrome），指的是社会、公司和组织中的一种文化现象，人们不愿意使用、购买或者接受某种产品、研究成果或者知识，不是出于技术或者法律等因素，而只是因为它源自其他地方（摘自http://zh.wikipedia.org/wiki/NIH综合征）。——译者注

它们，就可以方便地一窥HTTP请求的真面目了。

1. GET请求的工作原理

首先，请在终端窗口中执行以下命令启动nc并让它监听1156号端口（端口号任意，这里只是举个例子）：

```
$ nc -l -p 1156
```

运行起来之后，再打开一个窗口，用curl向本地的1156号端口发送一条GET请求：

```
$ curl http://localhost:1156/foo
```

请注意localhost:1156之后的/foo。切换至第一个终端窗口，你会看到nc打印出了以下内容：

```
GET /foo HTTP/1.1
User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3
➥OpenSSL/0.9.7l zlib/1.2.3
Host: localhost:1156
Accept: */*
```

这便是curl发送的HTTP请求报文。HTTP是一种文本协议，阅读和调试起来都很方便。报文的第一行给出了请求的类型（GET）、资源（/foo）以及当前会话所使用的协议（HTTP 1.1）。紧随第一行之后的是若干HTTP协议头，它们都是与当前请求相关的附加信息，其中一部分会影响服务器的行为。协议头以名称和:开头。所有协议头罗列完毕之后有一个空行，它标志着报文头部的结束，剩下的便是消息正文（如果有的话）。本例中的消息正文为空。

当你浏览网页时，Web浏览器发给服务器的请求通常都是GET请求——无论是点击链接、打开书签，还是在浏览器的地址栏手工输入URL，都是如此。在处理这种请求时，服务器会将页面内容作为应答发送给浏览器。

HTTP服务器发送的应答报文的第一行可分为三部分，首先是协议版本，紧接着是数值状态码，最后是一段原因说明，这段文本以人类可读的方式解释状态码（这段描述没有什么标准可言，也不一定非要用英语）。状态码的最高位数字用于表示状态的类别。例如在找不到客户端所请求的资源时，服务器的应答报文通常会以下述内容开始：

```
HTTP/1.1 404 Not Found
```

最高位的4表示服务器认为这个错误源自客户端（例如试图请求不存在的资源）。除第一行以外，应答报文的格式与请求报文相同：若干个协议头，紧跟着一个空行，然后是消息正文。如果网页请求有效，一般Web服务器会返回与下述内容相近的应答：

```
HTTP/1.1 200 OK
Date: Sat, 08 May 2010 19:09:55 GMT
Server: Apache
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
 <head>
  <title>Front Page</title>
 </head>
```

```
  <body>
   <h1>Welcome</h1>
  </body>
</html>
```

本例中的消息正文是一段HTML文档（参见协议头中的Content-Type字段）。根据资源的性质，HTTP请求可能会返回各种类型的数据，比如JPEG图片或PDF文档。Content-Type协议头可以帮助客户端区分应答中的数据类型。

毫无疑问，GET是最常见的请求类型。不过HTTP支持的请求类型可不止一种——而是8种，每种请求分别对应于一个动词（verb）——在后面的REST式接口的开发过程中，还要用到其中的PUT和DELETE动词。接着往下看。

2. PUT和DELETE

为了演示PUT的用法，请先创建一个名为put.txt的文本文件，文件内容只有一个单词Erlang：

```
$ echo Erlang > put.txt
```

接下来，请退出运行中的nc和curl会话，然后重启nc。下述命令中的-T选项用于让curl将文件PUT至指定的URL：

```
$ curl -T put.txt http://localhost:1156/foo
```

nc的输出如下：

```
PUT /foo HTTP/1.1
User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3
➥OpenSSL/0.9.7l zlib/1.2.3
Host: localhost:1156
Accept: */*
Content-Length: 7
Expect: 100-continue

Erlang
```

可以看出这段报文与之前的GET请求颇为类似。区别在于第一行中的GET变成了PUT，请求报文的末尾处多了两个协议头，用于表示报文头部完结的空行之后还有一段消息正文（即put.txt文件的内容）。Content-Length协议头的值为7，这是单词Erlang再加上一个（类UNIX系统上的）换行符的长度，这个换行符是在创建文件时由echo命令加上的。

发送完PUT请求后，如果仔细观察nc的输出，你可能会注意到请求报文中的消息正文并不是立即出现的。这是请求中的协议头Expect: 100-continue的缘故。Expect协议头可用于提升Web交互的效率。如果发现请求报文中有它，服务器就应该发送应答“100 Continue”，指示客户端继续发送消息正文。如果服务器根本不打算处理消息正文，它可以选择立即关闭连接，以免客户端辛辛苦苦传了半天数据，却落得个被服务器丢弃的下场。我们将在11.2.3节中实现自己的Web服务器，届时你将会看到这个协议头的处理办法。

最后，让我们来看看DELETE请求。再次重启nc，然后按照以下命令用curl发送DELETE请求：

```
$ curl -X DELETE http://localhost:1156/foo
```

不必担心——什么都删不掉；nc只会忠实地显示请求报文的内容：

```
DELETE /foo HTTP/1.1
User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3
➥OpenSSL/0.9.7l zlib/1.2.3
Host: localhost:1156
Accept: */*
```

可以看出，除了第一行中的动词有所不同以外，DELETE和GET的请求报文如出一辙。然而二者的语义却有着天壤之别：同样是针对资源/foo，这个请求可不是要下载，而是要让服务器删除该资源。请注意，我们不打算在此讨论资源的精确含义——这是个非常抽象的概念。对于一般的Web服务器来说，资源通常就是文件系统上的文件：GET表示“请给我发送一份该文件的副本”，PUT表示“文件上传”，DELETE表示“文件删除”。当然了，具体含义还得看Web服务器。就simple cache接口而言，将这些动词视作文件操作显然是不合时宜的，应当将它们转译为对应的缓存操作。

利用HTTP甚至可以实现一套比萨配送服务，这套服务中的GET表示订购比萨（可以将配送地址连同比萨的种类、数量等信息一并编码到URL中），PUT表示上传新食谱至菜单，DELETE表示删除食谱。HTTP规范只关注“资源”和“资源的表述形式”（如果将比萨和比萨订购分别视作资源和请求，那么送到门口的货真价实的比萨便可被视作资源的一种“表述形式”）。现在你已经初步掌握了HTTP协议，让我们开始深入Web服务器的实现细节吧。

11.2.2　实现一套通用的Web服务器行为模式

我们打算将这套初级的Web服务器实现成一个新的OTP行为模式，从而把它包装成独立的可复用组件。首先，请效仿11.1.2节中的tcp_interface，在现有应用的基础上搭建好新应用的架子。新应用的名字是gen_web_server，其目录结构如下：

```
gen_web_server
    |-- ebin
    |    `-- gen_web_server.app
    `-- src
         |-- gen_web_server.erl
         |-- gws_connection_sup.erl
         `-- gws_server.erl
```

gen_web_server与Erlang/OTP标准库stdlib一样，都属于库应用：这些应用本身无法启动，只能被用作其他主动应用的砖瓦。因此，.app文件中的mod项（用于指定应用启动入口）、application行为模式的实现模块，还有顶层监督者模块，统统都可以省略。

11

这个 Web 服务器非常简陋

这个 Web 服务器的功能并不完整，请不要把它用到线上环境中去！虽然为 REST 式接口的实现提供了一些便利，但它毕竟只是个示例程序，我们只是用它来演示自定义行为模式的创建方法和 Erlang/OTP 的实用编程实践而已。真正的 Web 服务器必备的数据分块、长连接等功能它都不具备。产品级质量的 Erlang Web 服务器有很多——如果需要，不妨去看看 Yaws 或 MochiWeb，Erlang/OTP 标准库自带的 inets httpd 服务器也不错。

在3.1.2节中我们曾经说过，行为模式可分为三部分：容器、接口和实现。现有的行为模式你都见识过了，使用它们时只需要提供行为模式的实现。这一次，你要打造的是一个全新的行为模式，它可能会有各式各样的实现，你的任务就是为这些实现提供接口和容器。在行为模式中，容器是可复用部分的主体，实现模块通过由接口定义的一组回调函数与容器对接。在本例中，这个容器就是我们的通用Web服务器。

gen_web_server容器的结构如图11-2所示。其中gen_web_server.erl是前端API模块，它负责屏蔽内部细节，行为模式的用户只会跟它打交道。行为模式的接口也由该模块指定。

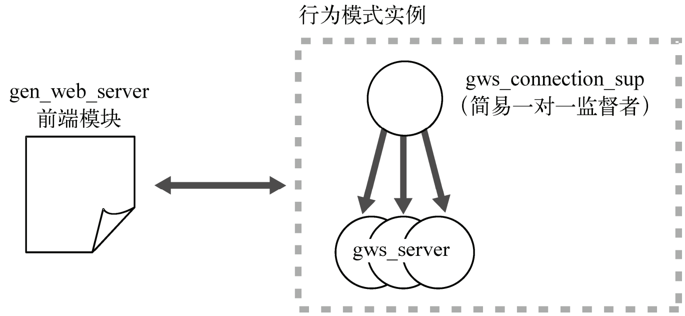


图11-2　gen_web_server的进程和模块

图11-1已经展示了gen_web_server容器实例的两个组成部分：一个监督进程，以及隶属于它的多个动态创建的服务器进程。

HTTP无非就是构筑于TCP之上的一个文本协议，因此gen_web_server实例的工作模式与11.1.3节中的tcp_interface应用大同小异。gen_web_server行为模式的实例也是由多个进程组成，进程数量可变而且没有上限。（相较之下，gen_server实例只有一个进程。）不同的实例负责不同的IP地址/TCP端口组合。在自己所负责的端口上，每个实例都可以处理为数众多的外部连接，其中每个连接都由一个独立的gws_server进程负责打理。

和tcp_interface应用一样，gws_connection_sup模块也是个简易一对一监督者，而且同样被用作处理器进程工厂。连接处理器由gws_server模块实现，该模块和11.1.3节中的ti_server类似；不过，虽然都是gen_server，HTTP协议的实现复杂度可比简单文本协议要大得多了。

我们先从前端模块gen_web_server讲起，你将会看到行为模式接口的定义非常简单明了。

1. gen_web_server模块：自定义行为模式

编译器只要看到`-behaviour(x)`声明，就会尝试调用模块x寻找行为模式接口。以ti_server为例，当编译器发现模块中写有`-behaviour(gen_server)`时，编译器便会调用`gen_server:behaviour_info(callbacks)`获取gen_server实现模块所应导出的回调函数列表。也就是说，行为模式应该与行为模式接口定义模块同名。

行为模式接口模块至少要导出一个函数，即`behaviour_info/1`。它只接受一个参数，即用于辨别待查询信息类别的原子。目前唯一合法的参数值是`callbacks`。对于其他参数值，该函数一律返回`undefined`。但如果碰到`callbacks`，它便会返回一张包含该接口所需的所有回调函

数的列表，其中每个函数以一个由函数名和元数组成的二元组表示。请在Erlang shell中调用gen_server:behaviour_info(callbacks)一探究竟，返回的列表很是眼熟吧：

```
1> gen_server:behaviour_info(callbacks).
[{init,1},
 {handle_call,3},
 {handle_cast,2},
 {handle_info,2},
 {terminate,2},
 {code_change,3}]
2>
```

在编写行为模式的实现模块时，一旦忘记实现（或导出）必要的函数，编译器便会根据这些信息发出警告。在定义新的行为模式时必须提供这个函数。代码清单11-5中罗列了gen_web_server模块的完整实现。如你所见，其中确实包含一个behaviour_info/1函数❶，该函数返回的列表中共有9个回调，除init/1以外，所有回调均与GET和POST等HTTP方法有关。

代码清单11-5 gen_web_server.erl

```
-module(gen_web_server).

%% API
-export([start_link/3, start_link/4,
         http_reply/1, http_reply/2, http_reply/3]).

-export([behaviour_info/1]).

behaviour_info(callbacks) ->
    [{init,1},
     {head, 3},
     {get, 3},
     {delete, 3},
     {options, 4},
     {post, 4},
     {put, 4},
     {trace, 4},
     {other_methods, 4}];
behaviour_info(_Other) ->
    undefined.

%%%===================================================================
%%% API

start_link(Callback, Port, UserArgs) ->
    start_link(Callback, undefined, Port, UserArgs).

start_link(Callback, IP, Port, UserArgs) ->
    gws_connection_sup:start_link(Callback, IP, Port, UserArgs).

http_reply(Code, Headers, Body) ->
    ContentBytes = iolist_to_binary(Body),
    Length = byte_size(ContentBytes),
    [io_lib:format("HTTP/1.1 ~s\r\n~sContent-Length: ~w\r\n\r\n",
                   [response(Code), headers(Headers), Length]),
     ContentBytes].
```

❶ 定义行为模式接口

❷ 启动新实例

❸ 构造HTTP应答

```
http_reply(Code) ->
    http_reply(Code, <<>>).

http_reply(Code, Body) ->
    http_reply(Code, [{"Content-Type", "text/html"}], Body).

%%%===================================================================
%%% Internal functions

headers([{Header, Text} | Hs]) ->
    [io_lib:format("~s: ~s\r\n", [Header, Text]) | headers(Hs)];
headers([]) ->
    [].

%% Fill in the missing status codes below if you want:
response(100) -> "100 Continue";
response(200) -> "200 OK";
response(404) -> "404 Not Found";
response(501) -> "501 Not Implemented";
response(Code) -> integer_to_list(Code).
```

实现gen_web_server行为模式的模块必须一个不落地导出❶中的所有9个回调函数；服务器会将收到的所有请求一一转发给实现模块中的相应回调。譬如收到的PUT请求将被转交给实现模块中的put/4函数处理。和gen_server不同，gen_web_server实现模块中的init/1函数负责的并不是整个行为模式的一次性初始化，而是图11-2中所有gws_connection连接处理器进程的初始化。最后，other_methods/4回调负责处理那些不太常用的HTTP方法，可用于实现WebDAV等HTTP扩展协议。

gen_web_server模块提供的API很简洁。一对start_link函数❷负责启动新的行为模式实例（其中start_link/3采用默认IP，start_link/4采用自定义IP——后者可用于拥有多个网络接口的主机）。除IP信息以外，start_link还需要一些参数，包括回调模块的模块名（每个行为模式都需要）、TCP监听端口号，以及一张附加参数列表。init/1会将这些附加参数传递给后续新建的所有连接处理器进程。这套API还提供了一个工具函数http_reply❸，用于辅助gen_web_server实现模块组织正确的HTTP应答报文；gws_server模块在生成“100 Continue”自动应答报文时也会调用这个函数。

搞定前端模块之后，下一步就是图11-2中的监督者模块gws_connection_sup。

2. gws_connection_sup模块

跟tcp_interface应用一样，gen_web_server也采用简易一对一监督者来管理连接处理器，这个监督者同样也兼任着连接处理器进程工厂的角色。每个gen_web_server实例都会在启动时创建一个gws_connection_sup进程来监听指定端口，只要服务器实例不退出，该进程就不会终止。

请将代码清单11-6中的gws_connection_sup与11.1.3节中的ti_sup模块（代码清单11-2）做一个对比，你会发现前者的start_link和start_child这两个函数多出了几个参数，而且监听套接字是在监督者模块的init/1函数中打开的。

代码清单11-6 gws_connection_sup.erl

```
-module(gws_connection_sup).

-behaviour(supervisor).

%% API
-export([start_link/4, start_child/1]).

%% Supervisor callbacks
-export([init/1]).

%%%===================================================================
%%% API functions

start_link(Callback, IP, Port, UserArgs) ->
    {ok, Pid} = supervisor:start_link(?MODULE, [Callback, IP,
                                               Port, UserArgs]),
    start_child(Pid),                                         <-- ❶ 启动第一个gws_
    {ok, Pid}.                                                    server子进程

start_child(Server) ->
    supervisor:start_child(Server, []).

%%%===================================================================
%%% Supervisor callbacks

init([Callback, IP, Port, UserArgs]) ->
    BasicSockOpts = [binary,                              ❷ 打开端口时用到的新选项
                     {active, false},
                     {packet, http_bin},
                     {reuseaddr, true}],
    SockOpts = case IP of
                   undefined -> BasicSockOpts;
                   _         -> [{ip,IP} | BasicSockOpts]
               end,
    {ok, LSock} = gen_tcp:listen(Port, SockOpts),
    Server = {gws_server, {gws_server, start_link,
                           [Callback, LSock, UserArgs]},
              temporary, brutal_kill, worker, [gws_server]},
    RestartStrategy = {simple_one_for_one, 1000, 3600},
    {ok, {RestartStrategy, [Server]}}.
```

本例中的监督进程运行起来之后，start_link/4函数会立即启动第一个gws_server子进程❶。这一步也可以并入gen_web_server:start_link/4中执行，不过当前这种做法可以确保新启动的gws_connection_sup进程背后总有一个进程随时待命，负责处理监听套接字上收到的外来连接。

打开监听套接字时，我们用到了一些新选项❷。首先，binary表示发送外来数据时应采用二进制串而非字符串的形式。其次，{active, false}表示用被动模式打开套接字。再次，{packet, http_bin}告诉套接字外来数据遵循HTTP格式。启用该选项后，套接字会自动将文本数据解析成更易于处理的消息，不但可以为你省去大量体力活，还可以提升HTTP请求的处理速度。详情参见稍后实现的gws_server模块。最后，{reuseaddr, true}用于快速复用本地端口号，如果不启用这个选项，服务器就必须等到监听套接字在OS内核中超时后才能再次（在同一端口上）启动。

TCP 流量控制与主/被动套接字

尽管主动模式条理更清晰，写出来的代码也更具 Erlang/OTP 风范，但它却无法提供流量控制。在主动模式下，套接字上有多少数据 Erlang 运行时系统就读多少数据，一读完就立即以 Erlang 消息的形式传递给持有套接字的进程。如果客户端的发送速度比接收方的读取速度快，那么消息队列就会不断增长并最终将内存耗尽。在被动模式下，持有套接字的进程必须显式读取套接字中的数据，这么做会增加代码的复杂度，但却可以更精细地控制系统接收数据的时机和速率，还可以依靠 TCP 内置的流量控制功能来自动限制发送方的发送速度。

你可能已经注意到了，在这个模块中我们打破了不给监督者添加额外功能的规矩。其原因在于gen_web_server是个库应用，没有可用于寄放这些代码的_app模块（tcp_interface应用是有的）。要想把这段代码挪出gws_connection_sup，至少还得再加一个进程，甚至很可能还得再增加一层监督结构。对于本书来说，未免太小题大作了。不过，按照这个思路对程序做个结构调整倒是个不错的练习。切记，只要服务器实例还活着，持有监听套接字的进程就不能停[①]。因此监听套接字不能在gen_web_server的`start_link/4`中打开，gws_server进程就更不用提了。

将这部分功能放入gws_connection_sup还有另外一个理由，那就是在实际应用当中，gws_connection_sup根本没有机会成为顶层监督者。如图11-3所示，无论何时，gen_web_server容器都只能在其他高层监督者的管辖之下作为某个应用的一部分而启动。

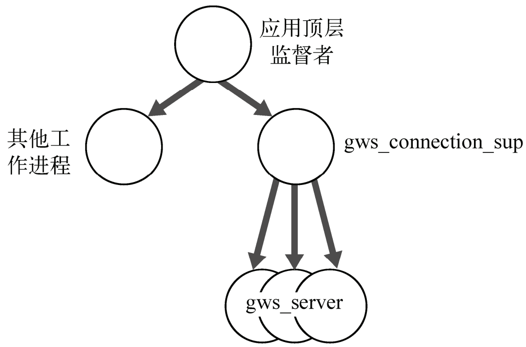


图11-3　在更大范围的应用中使用gen_web_server时的监督结构。gws_connection_sup不会被用作顶层监督者，即便代码中有bug，也不会殃及整个应用

也就是说，如果掺杂在`init/1`中的代码导致gws_connection_sup进程故障，上层监督者也有能力捕获这个错误。有了这重保障，即便故障会波及应用的其他部分，影响面也十分有限。不过gws_connection_sup要是重启失败，整个应用还是会彻底完蛋。这方面的风险不可小觑。

故障隔离与监督者的稳定性

图 11-3 中的顶层监督者一共启动了两个工作进程：左边的是个独立进程，右边的是 gen_

① 否则监听套接字将被自动关闭，参见11.1.3节。——译者注

web_server 的一个实例。监督者可以在协调多组进程的同时让它们彼此隔离。如果gws_connection_sup进程组发生崩溃，剩下的独立工作进程不会受到任何影响（不过监督者有可能会让它一并重启）。正是因为如此，我们才要求尽量不要在监督者中掺杂应用相关代码。

监督者通常都是高度可靠的。监督者一旦发生不测，整个监督体系的容错能力便会受到严峻挑战。顶层监督者尤其如此，它要是出了问题，整个应用都得跟着重启——如果是permanent 型应用（参见 10.1.3 节），甚至还会株连整个节点。

从实现策略上讲，gws_connection_sup和11.1.3节中的ti_sup之间的差异虽不明显，但却十分关键。明白了这一点，我们就可以开始研究更具实质性的问题了：那就是用于处理真正的HTTP连接的gws_server模块。

3. gws_server模块及{active, once}选项

和ti_server模块一样，gws_server也是用gen_server实现的。在套接字的处理策略上二者如出一辙，都采用了延迟初始化的手法。具体做法都是在init/1回调的返回值中将超时置零，从而将初始化延迟到handle_info(timeout,State)中。在初始化过程中，gws_server会阻塞等待gen_tcp:accept()返回，然后通知监督者派生新的处理器进程。不过在处理来自套接字的外部数据时，二者之间又存在明显的差异。

在tcp_interface应用中，我们曾经用过套接字选项{active, true}。启用该选项后，ERTS会自动读取套接字收到的所有数据并将之转换成Erlang消息发送给持有套接字的进程。这一机制有效简化了基于事件模型的代码，但它的缺点就是主动套接字无法提供流量控制。如果客户端的数据发送速度过快，服务器的内存就会被无节制增长的消息队列耗尽。另一方面，如果换用{active, false}选项，接收进程就必须通过gen_tcp:read()从套接字中显式读取就绪的数据。这样一来，如果接收方跟不上发送方的速度，TCP内置的流量控制机制便会生效，阻止发送方进一步发送数据，从而规避服务器端内存耗尽的风险。可是，这一方式又和Erlang的编程风格格格不入。

第三条出路就是{active, once}选项，这正是本例所采用的方法。该选项可以将套接字临时置为主动模式。等套接字再次收到数据，并以Erlang消息的形式将数据发送给持有者进程之后，套接字又会被自动重置为被动模式，TCP内置的流量控制也随之生效。在控制进程明确表态之前，套接字中的数据不会再被读取，也不会再产生新的消息。控制进程准备就绪后通常会再次启用{active, once}选项，然后等待下一条消息。下列代码中演示了如何配合简单的循环来使用{active, once}：

```
start() ->
    {ok, LSock} = gen_tcp:listen(1055, [binary, {active, false}]),
    {ok, Socket} = gen_tcp:accept(LSock),
    loop(Socket).

loop(Socket) ->
    inet:setopts(Socket, [{active,once}]),
    receive
        {tcp, Socket, Data} ->
```

11

```
            io:format("got ~p~n", [Data]),
            loop(Socket);
        {tcp_closed, _Socket} ->
            ok
    end.
```

第一步是创建用于接受外部连接的监听套接字，进而获取一个连接套接字。紧接着进入循环，在连接套接字上调用`inet:setopts/2`，启用`{active, once}`选项，然后坐等由套接字发出的消息。如有数据到来，就接收数据并进行处理。此时套接字又被重置成了被动模式。于是，在下一轮循环中，你需要再次启用`{active, once}`并等待新消息。

除上述差异之外，ti_server和gws_server模块之间最具实质性的区别还是在协议处理上。前者只实现了一套简单协议，后者却需要处理HTTP的子集。该模块的代码如代码清单11-7所示。乍一看代码很多，其实并不复杂。

代码清单11-7　gws_server.erl

```
-module(gws_server).

-behaviour(gen_server).

%% API
-export([start_link/3]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {lsock, socket, request_line, headers = [],          ❶ 服务器状态记录
                body = <<>>, content_remaining = 0,
                callback, user_data, parent}).

%%%===================================================================
%%% API

start_link(Callback, LSock, UserArgs) ->
    gen_server:start_link(?MODULE,
                          [Callback, LSock, UserArgs, self()], []).

%%%===================================================================
%%% gen_server callbacks

init([Callback, LSock, UserArgs, Parent]) ->
    {ok, UserData} = Callback:init(UserArgs),
    State = #state{lsock = LSock, callback = Callback,
                   user_data = UserData, parent = Parent},
    {ok, State, 0}.

handle_call(_Request, _From, State) ->
    {reply, ok, State}.
                                                                  处理请求报文 ❷
handle_cast(_Request, State) ->
    {noreply, State}.

handle_info({http, _Sock, {http_request, _, _, _}=Request}, State) ->
    inet:setopts(State#state.socket, [{active,once}]),
    {noreply, State#state{request_line = Request}};
```

```
handle_info({http, _Sock, {http_header, _, Name, _, Value}}, State) ->
    inet:setopts(State#state.socket, [{active,once}]),
    {noreply, header(Name, Value, State)};
handle_info({http, _Sock, http_eoh},
            #state{content_remaining = 0} = State) ->
    {stop, normal, handle_http_request(State)};
handle_info({http, _Sock, http_eoh}, State) ->
    inet:setopts(State#state.socket, [{active,once}, {packet, raw}]),
    {noreply, State};
handle_info({tcp, _Sock, Data}, State) when is_binary(Data) ->
    ContentRem = State#state.content_remaining - byte_size(Data),
    Body       = list_to_binary([State#state.body, Data]),
    NewState = State#state{body = Body,
                           content_remaining = ContentRem},
    if ContentRem > 0 ->
            inet:setopts(State#state.socket, [{active,once}]),
            {noreply, NewState};
       true ->
            {stop, normal, handle_http_request(NewState)}
    end;
handle_info({tcp_closed, _Sock}, State) ->
    {stop, normal, State};
handle_info(timeout, #state{lsock = LSock, parent = Parent} = State) ->
    {ok, Socket} = gen_tcp:accept(LSock),
    gws_connection_sup:start_child(Parent),
    inet:setopts(Socket,[{active,once}]),
    {noreply, State#state{socket = Socket}}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
%%%===================================================================
%%% Internal functions

header('Content-Length' = Name, Value, State) ->
    ContentLength = list_to_integer(binary_to_list(Value)),
    State#state{content_remaining = ContentLength,
                headers = [{Name, Value} | State#state.headers]};
header(<<"Expect">> = Name, <<"100-continue">> = Value, State) ->
    gen_tcp:send(State#state.socket, gen_web_server:http_reply(100)),
    State#state{headers = [{Name, Value} | State#state.headers]};
header(Name, Value, State) ->
    State#state{headers = [{Name, Value} | State#state.headers]}.

handle_http_request(#state{callback     = Callback,
                           request_line = Request,
                           headers      = Headers,
                           body         = Body,
                           user_data    = UserData} = State) ->
    {http_request, Method, _, _} = Request,
    Reply = dispatch(Method, Request, Headers, Body,
                     Callback, UserData),
    gen_tcp:send(State#state.socket, Reply),
    State.
```

❸ 处理协议头

❹ 报文头部结束，消息正文为空

❺ 报文头部结束，准备接收消息正文

❻ 等待连接，然后启动新的处理器

❼ 记住内容长度，为后续处理做准备

❽ 让客户端继续

❾ 执行回调获取结果

```
dispatch('GET', Request, Headers, _Body, Callback, UserData) ->
    Callback:get(Request, Headers, UserData);
dispatch('DELETE', Request, Headers, _Body, Callback, UserData) ->
    Callback:delete(Request, Headers, UserData);
dispatch('HEAD', Request, Headers, _Body, Callback, UserData) ->
    Callback:head(Request, Headers, UserData);
dispatch('POST', Request, Headers, Body, Callback, UserData) ->
    Callback:post(Request, Headers, Body, UserData);
dispatch('PUT', Request, Headers, Body, Callback, UserData) ->
    Callback:put(Request, Headers, Body, UserData);
dispatch('TRACE', Request, Headers, Body, Callback, UserData) ->
    Callback:trace(Request, Headers, Body, UserData);
dispatch('OPTIONS', Request, Headers, Body, Callback, UserData) ->
    Callback:options(Request, Headers, Body, UserData);
dispatch(_Other, Request, Headers, Body, Callback, UserData) ->
    Callback:other_methods(Request, Headers, Body, UserData).
```

这个gen_server的服务器状态记录中存的东西还真是不少❶。lsock和socket字段分别是监听套接字和连接套接字。request_line、headers、body和content_remaining字段用于处理HTTP协议。callback字段是行为模式实现模块的模块名，user_data则是需要传递给回调模块的应用相关数据。最后，parent字段中保存的是gws_connection_sup监督者的进程ID。

tcp_interface应用的ti_sup监督者模块是以注册进程的形式运行的，最多只能运行一个实例。但gen_web_server可以并行启动多个实例，为此gws_server进程必须知道自己隶属于哪个监督者。该信息由start_link/3函数负责采集，它会记下调用进程（一般情况下应该是某个gws_connection_sup进程）的pid，再以调用参数的形式自动传递给init/1。

start_link/3函数的参数包括行为模式回调模块、监听套接字以及一张附加参数列表（UserArgs）；这些附加参数由gen_web_server行为模式的用户提供，最终会原封不动地传递至init/1。UserArgs参数的传递构成了容器框架与用户提供的行为模式回调模块之间的首次交互（这和gen_server新实例启动时调用用户提供的init/1回调函数是一样的道理）。

init/1函数一返回，新的服务器进程便会触发超时，转而进入handle_info(timeout,...)。和此前一样，服务器将在此处阻塞执行accept()并要求启动新的处理器进程❻。请注意，正如代码清单11-7之前的简短示例所示，连接套接字立即被打上了{active, once}标记，进而被存入服务器状态记录。

搞定上述事项之后，剩下的便是HTTP协议的处理。请回想一下11.2.1节中的那些HTTP请求。以PUT请求为例，当消息正文为“Hello!”时，请求报文如下：

```
PUT /foo HTTP/1.1
User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3
➥OpenSSL/0.9.7l zlib/1.2.3
Host: localhost:1156
Accept: */*
Content-Length: 7
Expect: 100-continue

Hello!
```

一般而言，HTTP请求报文的结构如下：

- ❑ 请求行；
- ❑ 协议头（数量任意）；
- ❑ 一个空行；
- ❑ 消息正文。

还记得我们在代码清单11-6中创建监听套接字时用到的{packet, http_bin}选项吗？只要启用该选项，套接字就会替你完成协议解析工作，发出的消息也更易于处理。首先，当新请求到来时，请求行将被解析为以下格式的消息（沿用上述的PUT示例）：

```
{http, Socket, {http_request, 'PUT', <<"/foo">>, {1,1}}}
```

请注意，HTTP方法是以原子的形式给出的（位于单引号内）。只有HTTP规范中列出的7个常用方法才享有这个待遇。对于其他无法识别的HTTP方法，方法名会以二进制串的形式给出——例如<<"PATCH">>，或是<<"MKCOL">>（对应于WebDAV中的MKCOL方法）。

收到的请求行❷会被存入服务器状态。然后，你应该再次将套接字置为{active, once}以读取更多客户端数据。可以看到，在需要从套接字上读取更多数据时，handle_info函数的各个子句都会这么做。

处理完请求行之后，一般还会收到若干协议头。启用{packet, http_bin}选项之后，套接字会自动解析各个HTTP协议头，然后按以下格式给你发送消息：

```
{http, _Socket, {http_header, Length, Name, _ReservedField, Value}}
```

负责处理这些消息的handle_info子句❸将大部分实质性工作委托给了内部函数header/3。该函数主要负责将各协议头存入服务器状态中的一个列表内，不过在两种特殊情况下还有些额外工作要做。在处理Content-Length协议头时❼，数据要以整数形式存入服务器状态的content_remaining字段。这份数据将用于接收请求报文的消息正文。碰到携带"100-continue"的Expect协议头时❽，应向客户端发送HTTP应答"100 Continue"以告知客户端继续发送数据（参见11.2.1节）。如果不发送应答，客户端会先暂停两秒钟再开始传输消息正文，从而大大拖慢整体进度。

遇到用作请求报文头部结束符的空行时，套接字会发送一条以下格式的消息：

```
{http, _Socket, http_eoh}
```

如果Content-Length协议头的值为零（保存在服务器状态记录的content_remaining字段中），就意味着请求报文不含消息正文❹。这时你只要处理请求并发送应答就可以了。但如果消息正文非空❺，则需要将套接字从{packet, http_bin}状态切换至{packet, raw}状态，让它停止HTTP解析，把接收到的数据转换成普通的{tcp, Socket, Data}格式。收到这类消息时，必须将数据追加到服务器状态的body字段（初始为一个空的二进制串）并相应减小content_remaining计数，直至该计数器归零。直到那时你才能够用handle_http_request/1处理完整的请求报文。

gen_web_server容器代码与用户定义的行为模式实现模块（即回调模块）之间的交互主要发

生在这个函数中。此前我们只在设置服务器状态记录的`user_data`字段时调用过`init/1`回调。现在，你得按消息中注明的HTTP方法将之前采集到的所有与请求报文相关的信息一一派发给相应的回调函数。

为此❾，首先需要从服务器状态中提取所需的字段，包括请求结构中的HTTP方法名。然后将请求派发给回调模块中的对应函数，由这些函数负责构造HTTP应答报文。最后，只要通过套接字将应答报文发送至客户端就万事大吉了！

哇！这段路真是跑得气喘吁吁，希望你还能跟得上我们的脚步。回顾一下一路攒下的代码，还不算赖——毕竟这还只是一个最为简陋的HTTP服务器。

不过这些辛苦都是值得的：有了gen_web_server行为模式，给缓存应用添加HTTP接口可不就是小菜一碟儿了嘛！毕竟这才是我们自11.2节以来的主要目标。只需要创建一个gen_web_server行为模式实现模块，再定义几个负责与缓存应用进行交互的回调函数就功德圆满了。不过首先，我们不妨先来看看REST到底是什么意思，REST式的接口又是什么样子的。

11.2.3　初识REST

REST即表征状态转移（representational state transfer）。这个概念总结了HTTP中的若干已成为既成事实的核心思想。这些思想并不局限于HTTP，但HTTP的支持极为广泛，而且还提供了`GET`、`PUT`、`POST`和`DELETE`等一整套为人所熟知的状态处理操作和状态迁移操作。不费吹灰之力我们就可以将这几个动词映射成数据库的CRUD操作，这就大大简化了HTTP缓存接口的设计。

表征状态转移

REST 的关键原则是资源由服务器持有，每个资源都有一个对应的全局标识符（URI），客户端采用标准接口与资源的表述形式（文档）进行协作。也就是说，客户端获取到的只是资源的表述形式，而非资源本身。所谓资源，可能只是一个抽象概念，比如“此刻我窗外的景色”。我们可以用各种格式来表述资源（比如 JPEG、PNG 或 TIFF 等）。每个请求都会导致客户端状态的迁移，这和你在网站上点击浏览网页是一个道理。文档内通常还会包含指向其他资源的标识符，比如指向其他 HTML 页面的链接。

这里有一个要点必须注意，那就是服务器不应该在自身状态中保存任何与客户端相关的隐含信息：状态信息要么保存在客户端，要么以可寻址资源的形式显式保存在服务器端。有些人可能会觉得不管是什么遗留系统，只要套上一层 HTTP 接口就能自封为 REST 风格，这最后一条原则彻底否定了这种想法。

REST式接口与与Simple Cache的交互很简单，只会用到3个HTTP动作：`GET`、`PUT`和`DELETE`。不过你必须确切定义每个动作分别对应于哪个缓存操作。最简单的就是`DELETE`，提供资源名称即可，无须另行收发其他数据：

```
DELETE /key HTTP/1.1
```

这一请求最终会调用`simple_cache:delete(Key)`，从缓存中清除由请求报文指定的键。该请

求的应答永远都是"200 OK"，且应答报文不含消息正文。

GET要略为复杂一些：

```
GET /key HTTP/1.1
```

处理这一请求时需要调用simple_cache:lookup(Key)完成查找操作。如果找不到，就回复"404 Not Found"，消息正文为空；如果找到了，就回复"200 OK"，消息正文就是从缓存中查到的值（需要转换为纯文本格式）。

最后是PUT。请求报文的消息正文中包含需要存入缓存的值：

```
PUT /key HTTP/1.1
```

处理这类请求时，需要调用simple_cache:insert(Key, Body)。请求的应答永远都是"200 OK"，且消息正文为空。

就这么多了。对于这样一个简单应用，很容易就可以看出它的接口的确符合REST的原则：每个资源（缓存中的条目）都有自己的URL，只用基本的HTTP操作来操纵资源，整套接口也不涉及任何跨请求的客户端状态信息。下一节，我们将以先前搭建的gen_web_server为基础来实现这套协议。

11.2.4 用gen_web_server实现REST式协议

和11.1.2节中的tcp_interface一样，我们的REST式HTTP接口http_interface也是一个独立的主动应用，带有自己的application行为模式模块和顶层监督者。监督者只有一件事情要做：启动一个gen_web_server行为模式容器的实例。应用的目录结构如下：

```
http_interface
    |-- ebin
    |   `-- http_interface.app
    `-- src
        |-- hi_app.erl
        |-- hi_server.erl
        `-- hi_sup.erl
```

对你来说，编写.app文件、_app模块还有_sup模块应该早就不是问题了。重点在于_sup模块的监督策略，此处采用的是普通一对一监督（而不是简易一对一）；而且hi_server:start_link/1函数只启动了一个子进程，该进程还是permanent型（而不是temporary型）。注意，hi_server模块对外屏蔽了应用内部的实现细节，只有它才知道内部的功能逻辑是基于gen_web_server实现的。

为了简化端口号的配置，可以仿效代码清单11-1中的ti_app模块，在hi_app模块的start/2函数中尝试用application:get_env/2来读取端口号（不过套接字可不能在这儿打开）。如果找不到对应的配置项，请采用默认端口号。为了避免与tcp_interface的1155号端口相冲突，这次我们选用1156作为默认端口号。（愿意的话，你也可以采用这种方法来配置IP地址；当然，简单起见，暂时采用默认IP地址也没问题。）我们相信，你一定能够凭借自己的能力完成这些工作。

该应用的结构如图11-4所示。此处最为关键的模块就是hi_server，正是它基于gen_web_server行为模式实现了真正的REST式HTTP接口。该模块的代码参见代码清单11-8。

11

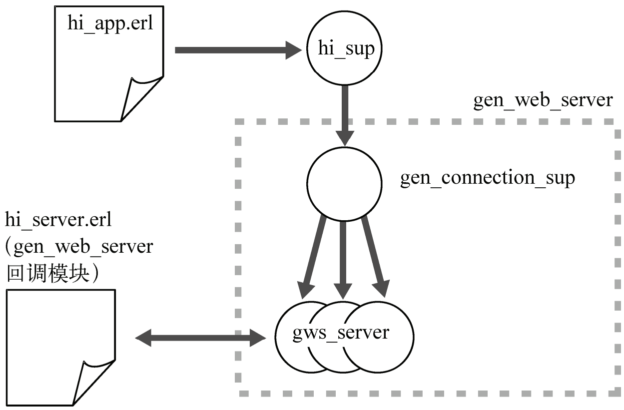


图11-4　REST式接口应用的结构。顶层监督者hi_sup（由hi_app启动）只有一个。该进程与其自身的子进程一起，构成了一个以hi_server为回调模块的gen_web_server行为模式的实例

代码清单11-8　hi_server.erl

```
-module(hi_server).

-behaviour(gen_web_server).

%% API
-export([start_link/1, start_link/2]).

%% gen_web_server callbacks
-export([init/1, get/3, delete/3, put/4, post/4,
         head/3, options/4, trace/4, other_methods/4]).

%%%===================================================================
%%% API

start_link(Port) ->
    gen_web_server:start_link(?MODULE, Port, []).

start_link(IP, Port) ->
    gen_web_server:start_link(?MODULE, IP, Port, []).

%%%===================================================================
%%% gen_web_server callbacks

init([]) ->
    {ok, []}.

get({http_request, 'GET', {abs_path, <<"/",Key/bytes>>}, _},
    _Head, _UserData) ->                                          ❶ 处理GET请求
    case simple_cache:lookup(Key) of
        {ok, Value} ->
            gen_web_server:http_reply(200, [], Value);
        {error, not_found} ->
            gen_web_server:http_reply(404, "Sorry, no such key.")
    end.
```

```
delete({http_request, 'DELETE', {abs_path, <<"/",Key/bytes>>}, _},
        _Head, _UserData) ->
    simple_cache:delete(Key),
    gen_web_server:http_reply(200).
put({http_request, 'PUT', {abs_path, <<"/",Key/bytes>>}, _},
    _Head, Body, _UserData) ->
    simple_cache:insert(Key, Body),
    gen_web_server:http_reply(200).

post(_Request, _Head, _Body, _UserData) ->
    gen_web_server:http_reply(501).

head(_Request, _Head, _UserData) ->
    gen_web_server:http_reply(501).

options(_Request, _Head, _Body, _UserData) ->
    gen_web_server:http_reply(501).

trace(_Request, _Head, _Body, _UserData) ->
    gen_web_server:http_reply(501).

other_methods(_Request, _Head, _Body, _UserData) ->
    gen_web_server:http_reply(501).
```

❷ 处理DELETE请求

❸ 处理PUT请求

❹ 其余方法一律回复"501 Not Implemented"

大功告成！在gen_web_server行为模式上耗费的精力终于有了回报：有了它，实现这类HTTP服务器简直不费吹灰之力。和gen_server实现模块类似，hi_server同样具备behaviour声明、回调函数导出表，以及若干用于启动服务器实例的API函数。在本例中，两个start_link函数需要一个参数来指明端口号，该参数传自hi_sup。同时请注意，这两个函数还向gen_web_server:start_link()传递了一个用作UserArgs参数的空列表；就这个服务器实现而言，这个参数派不上用场。

每当有新连接到来时，gen_web_server就会调用init/1回调来初始化user_data字段。用户传给gen_web_server:start_link()的UserArgs参数会被原封不动地传递至init/1，因此这个服务器的输入参数就是一张空表。不过这个简单的服务器根本就没有用到gen_web_server的user_data功能，因此init/1返回的也是一张空表。

在整套协议实现之中，最为关键的还是get、delete、put这3个回调。3个函数利用二进制模式匹配（参见2.10.2节）提取出了URI中的键（去掉开头的斜杠即可）。简洁起见，本例中的键和值仅限于内容为纯文本的二进制串。虽然缓存中也可以存储其他类型的Erlang项式，但该接口不予支持；尤其需要注意的是，当simple_cache:lookup(Key)调用成功时，查得的值也必须是二进制串、字符串或IO列表，否则服务器无法通过套接字发送应答。

在处理GET请求时❶，如果能在缓存中查到给定的键，查得的值便会作为"200 OK"应答的消息正文返回给客户端；否则就返回消息正文为空的"404 Not Found"应答。DELETE请求❷比较简单：删完给定的键后回复"200 OK"即可。PUT请求❸也复杂不到哪儿去，它的作用是将URI中的键和请求消息正文中的值存入缓存。（请注意，put回调需要Body参数，get和delete则不需要。）至于其余的HTTP方法❹，统统返回"501 Not Implemented"。

我们还可以给应答报文添加更为丰富的协议头，不过HTTP协议头使用规范的详情已经超出

了本书的讨论范畴。需要（或应该）包含哪些附加信息，很大程度上是取决于应用场景的。

来试试Simple Cache的HTTP接口吧。首先请采用以下命令编译源码。为了让编译器检查hi_server模块是否正确实现了gen_web_server行为模式的接口，我们给erlc命令加上了gen_web_server的.beam文件路径：

```
erlc -pa ./gen_web_server/ebin -o ./http_interface/ebin
➥./http_interface/src/*.erl
```

请按11.1.5节中给出的方法启动系统（还可以同时启动tcp_interface应用），然后在系统shell中调用application:start(http_interface)。操作完毕后，可以用Appmon检查应用的运行状况。

首先，利用curl发起PUT请求，以xyzzy为键将文件put.txt（创建于11.2.1节，文件内容为单词Erlang）上传至缓存服务器，然后再用GET请求查询同一个键：

```
$ curl -T put.txt http://localhost:1156/xyzzy
$ curl http://localhost:1156/xyzzy
Erlang
```

确实管用！不妨用TCP接口查询同一个键试试看。不过由于HTTP接口只认得二进制串，这次得换用二进制串来表示：

```
lookup[<<"xyzzy">>]
OK:{ok,<<"Erlang\n">>}.
```

如果想改进这套接口，可以考虑从存取任意Erlang数据入手。另外，在处理PUT请求时，不但可以保存消息正文，还可以在缓存中保存消息正文的content-type。这样一来就可以给GET请求的应答报文加上对应的content-type了。

11.3　小结

现在外部的客户端可以通过两种不同的协议与Simple Cache应用集成：一种是自定义的、基于TCP的文本协议，一种是基于标准HTTP的更为结构化的REST式协议。这一路上，我们不仅掌握了用Erlang开发并发TCP服务器的若干关键技能，还学会了如何创建自定义行为模式，顺便还学了一点儿HTTP。如今，生产环境中多语言、多平台共存的情况日益普遍，你在本章中作出的努力成功拓展了Simple Cache的应用场景，把它推向了Erlang以外的世界。

下一章，我们将会展示让Erlang直接与其他语言开发的程序进行交互的方法，从而达成比TCP/IP更为紧密的集成。

第12章

用端口和NIF集成外围代码

本章概要

- 端口、链入式驱动，和原生函数（Natively Implemented Functions，NIF）
- 集成第三方C库
- 用NIF集成C库

现如今，我们的缓存应用已经十分强大了，日志、多节点分布式支持、自动化集群接入，等等，统统不在话下。它不仅提供了用于存取二进制数据的REST式HTTP接口，还提供了较为简陋但却可以存取任意Erlang项式的TCP文本接口。总体来讲，Erlware的同仁们颇为满意。再接再厉，如果能通过HTTP接口来存取结构化数据，岂不更妙？当然，我们应该采用标准的数据格式，而且最好不要让客户端操心序列化、反序列化等事务。Erlware希望你能进一步扩展现有的REST式API，使它能够支持JSON（JavaScript Object Notation，JavaScript对象标记）格式的数据存取。（详情请参见www.json.org。另外请注意，为了与YAJL中的术语保持一致，在本章中我们统一将JSON对象称作映射。）

他们打算采用名为YAJL的开源JSON库来实现JSON文本和Erlang项式间的相互转换。YAJL库用C写成，是类似于SAX的回调式解析器。利用它，无须借助任何中间格式就能把JSON字符串解析成所需的Erlang项式。最新版本的YAJL可从http://lloyd.github.com/yajl/获取。还可以在GitHub.com上找到一个可与本书其余代码配套使用的版本（请到GitHub上搜索*Erlang and OTP in Action*）。

之所以选择YAJL，是因为我们认为解析JSON这样的标准格式最好还是选用久经考验的库，浪费精力自己造轮子不值得。另外，与C库进行交互也有一定的开销，小段JSON文档的解析并不比纯Erlang快多少。恐怕只有在解析大型文档（比如内含数兆字节Base64编码数据的文档）时，这个方案才能发挥出优势。同时，我们还会介绍Erlang与外界交互的通用机制。

与大部分编程语言一样，Erlang也允许你与其他语言写成的代码进行交互，不过它提供的标准机制着实有点儿特别。大部分语言都自带一套外围函数接口（Foreign Function Interface，简称FFI），用于在宿主语言中链接和调用C代码。Erlang则别出心裁地扩展了消息传递机制，拿它来实现与外围代码的交互。在Erlang代码看来，外围代码的行为和独立的Erlang进程没有什么两样：外围代码同样可以收发Erlang消息。外围代码在Erlang中以端口的形式出现。所谓**端口**，就是一

种类似于Erlang进程的对象（参见2.2.7节）。

除此以外，还可以利用Erlang的分布式机制来集成外围代码。Erlang/OTP自带两个库，分别是C库Erl_Interface（简称为ei）和Java库Jinterface——利用这两个库可以写出能够乔装成Erlang节点的程序。对于真正的Erlang节点（即运行在分布式模式下的Erlang VM）而言，这些外围程序和别的节点没有什么区别——它们有节点名，而且也都遵循Erlang的分布式通信协议。不过JSON文档解析这类任务还用不着创建C节点（或Java节点），否则未免也太兴师动众了。我们将在第13章中详细介绍用Jinterface创建外围节点的方法。

最后一种集成手段就是原生函数（NIF），和本书一样，它也才刚刚诞生不久。利用NIF可以创建出类似于Erlang内置函数（BIF）的函数。这些函数各自隶属于某个特定的Erlang模块，虽然用C和erl_nif库开发而成，它们的调用方法却和普通Erlang函数一般无二。在3种集成手段之中，NIF的通信开销最小；然而NIF代码一旦出现问题，整个Erlang VM都有可能会跟着宕机。因此千万不要滥用NIF，使用之前务必要好好掂量一下。下面我们先介绍端口，然后再介绍NIF。

12.1　端口和 NIF

在Erlang的各种对外交互方式中，端口是最为传统也最为基本的一种。其中又以普通端口最为简单优雅，同时它还在Erlang代码和外围代码之间筑起了一道关键的隔离带，性能方面一般情况下也还过得去。此外，端口与开发语言无关：任何语言都可以用于端口外围程序的开发。如果拿不准该采用哪种方式，那么不妨先用端口编写一个简单版本，待后续发现需要提升性能时再做优化也不迟。

端口以消息传递为基本通信形式。要想在Erlang中向端口背后的外围代码发送数据，请按以下格式向端口发送消息：

```
PortID ! {self(), {command, Data}}
```

其中`Data`既可以是二进制串也可以是IO列表（即多层嵌套的字节列表和/或二进制串列表）。请注意，消息中必须带上端口属主进程的pid，属主进程通常就是当前进程，但其实只要知道了端口ID和属主进程的pid，任何进程都可以给端口发消息。（erlang模块还提供了一批可以越过所有权直接操控端口的BIF函数，但在此我们只讨论消息传递方式。）

在向Erlang发送源自外围代码的数据时，端口会将数据封装成以下格式的消息，异步发送给端口的属主进程：

```
{PortID, {data, Data}}
```

创建端口时可以指定多种选项。选项不同，`Data`字段的格式也会有所不同。常见的如二进制串或字节列表。此外，还可以让端口按固定大小分块发送数据，采用纯文本协议时则可以选择按行发送数据。

端口分为两种：其中*普通端口*执行的外围代码都是运行于Erlang VM之外的外部程序。这些外部程序都是独立的操作系统进程，通过标准输入和标准输出与Erlang交互。包括shell脚本在内

的任何程序，只要能够在宿主操作系统下运行，并采用标准输入和标准输出来完成任务，都可以被普通端口调用。

以下示例可在Erlang中运行操作系统命令echo 'Hello world!'：

```
Port = open_port({spawn, "echo 'Hello world!'"}, []).
```

（打开端口的方法留待12.2节讨论。）端口发回的数据如下：

```
{#Port<0.512>, {data,"'Hello world!'\n"}}
```

有些可执行程序无法直接通过标准输入/输出和Erlang交互，这时就必须通过shell脚本或C程序来进行适配。不过大部分UNIX程序可以直接为Erlang所用。

在使用普通端口时，最坏的情况无非也就是外围程序崩溃导致端口关闭；Erlang VM丝毫不会受到影响。在这种情况下，Erlang代码完全可以检测到异常并采取相应的措施，比如重启外围程序等。

另一种端口采用的是链入式驱动，通常叫做*端口驱动*。顾名思义，*链入式驱动*是个可被Erlang VM动态加载和链接的共享库，通常用C开发而成。它的优点是通信效率高；但和NIF一样，链入式驱动有可能导致整个Erlang VM的崩溃。和普通端口一样，链入式驱动也以字节为单位进行通信；站在Erlang代码的角度看，两种端口完全一样。因此，如果普通端口不够用，将之升级成链入式驱动可谓易如反掌。

这两种端口便是本章的主题，我们将逐一介绍它们的使用方法和应用场景。只有掌握了它们的优缺点，你才能够在自己的系统环境下作出正确的选择。

12.1.1 普通端口

在Erlang中，端口是最简单也最为常见的外围代码通信手段。这类对象脚踏两条船：一只脚踩着Erlang，另一只脚踩着操作系统。在Erlang这边，端口和Erlang进程差不多：先创建再使用，利用普通消息传递进行通信，最终又会在某个时刻消亡。创建出的每个端口都带有一个永不回收的唯一标识符。端口本身无法执行任何Erlang代码，但每个端口都隶属于一个Erlang进程，它就是这个端口的*属主*。端口会把从外界收到的数据发送给属主，数据的处理方式由属主决定。打开端口的进程将成为端口默认的属主，利用BIF erlang:port_connect/2可以将所有权转移给其他进程。属主进程一旦终止，端口便会随之自动关闭。

从操作系统的角度来看，普通端口无非就是运行在操作系统中的另一个程序，只是标准输入和标准输出与Erlang VM对接在了一起而已。即便程序崩溃，Erlang也检测得到，并且随时可以采取重启等措施。外围程序在独立的地址空间内运行，只能通过标准I/O与Erlang交互。因此无论外围程序如何“撒野”，运行中的Erlang系统都不会受其影响而崩溃。考虑到外围代码潜在的危险性，这无疑是一个巨大的优势，同时也是Erlang系统能够在与其他程序进行交互的同时保持高度稳定的原因之一。如果实时性要求不高，硬件驱动甚至也可以如法炮制。外围代码与Erlang的对接方式如图12-1所示。

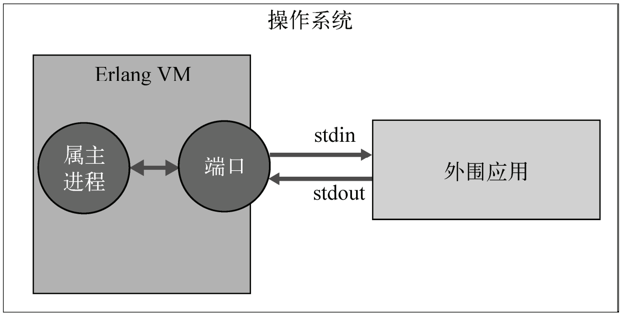


图12-1 Erlang通过普通端口的标准输入和标准输出与外围代码进行通信。外围代码一旦崩溃，端口就会关闭

当然，天下没有免费的午餐。安全背后的代价是性能。数据必须以字节流的形式往返于两个进程之间。为此你得定义一套字节流协议，并自行完成数据的序列化和反序列化。需要传递复杂数据时，可以利用Erl_Interface ei库中的函数，但前提是必须用C作为开发语言。与ei类似，Jinterface库为Java程序提供了Erlang通信支持。至于其他语言嘛，就只能自力更生了，或许Elang/OTP自带的IDL编译器（IC）可以派上一些用场。当然，应用需求不同，实现策略也可繁可简。以开关控制程序为例，用1表示开、用0表示关，定义一套字符协议就可以了。

12.1.2 链入式端口驱动

从表面上看，链入式驱动和普通端口一模一样：外围代码与Erlang代码之间的通信也是通过面向字节的消息传递来完成的。然而二者在底层消息传递机制层面却完全不同。而且链入式驱动运行于Erlang VM之内，二者共享一个操作系统进程空间。要问个中缘由，主要还是为了性能。

这样做也有负作用——而且还相当严重——端口驱动一旦崩溃，整个Erlang系统都会宕机。更要命的是，用于开发端口驱动的C语言，偏偏又将大量错误检测和资源管理的工作抛给了开发者。开发者们都是肉体凡胎，难以妥善地处理这些问题。一般来说，用C或更为低级的语言开发的代码在稳定性上都要略逊一筹；偏偏这些不够稳定的代码又会直接链入Erlang VM，任何错误都有可能导致整个Erlang VM宕机，碰上这种情况，Erlang中的任何容错手段都回天乏术了。链入式端口驱动的工作方式参见图12-2（不妨与图12-1做一下对比）。

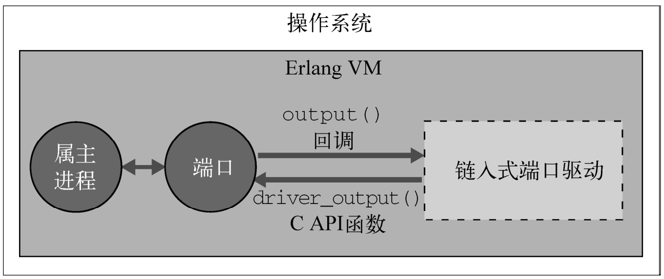


图12-2 链入式驱动和Erlang VM在同一操作系统进程空间内运行，二者通过C API中的回调函数和缓冲区完成数据传输

总之请谨记一点，使用链入式驱动就是在拿安全换速度。除非系统的确对速度有所苛求，否则不要轻易做出妥协。详情请参见第14章中有关性能评测和优化的讨论。

12.1.3 原生函数（NIF）

NIF是Erlang中的新特性。虽然在各种外围程序通信方案中，端口最为简单、安全，但有时候我们还是忍不住去追求通信开销的最小化，最好能做到像内置函数一样快。于是，NIF闪亮登场。它可以帮你把C函数变成跟BIF一样的Erlang函数。这种方式跟Java、Python等其他语言的外围函数接口差不多，用过这些语言的话，应该就不会感到陌生了。不要会错意，NIF绝不应该成为你的第一选择——想象一下，一套复杂系统说崩溃就崩溃，除了核心转储文件以外什么线索也没留下，那该是多么恐怖的一件事情。如果确定要用NIF，最好先把系统中的关键节点与运行NIF代码的节点隔离开来，直至NIF代码在实战中百炼成钢为止。

NIF的另一个根本问题在于：原生函数是在调用该函数的VM线程上下文内运行的，NIF不将控制权交还给VM，VM就无法再次调度该线程。这个问题导致NIF只适用于那些能够迅速返回的函数；长期运行的NIF会阻塞Erlang VM的调度器。

总结一下，供Erlang与其他语言进行交互的底层机制一共有3种：普通端口、链入式端口驱动以及NIF。三者各有各的优势：普通端口既安全又简单，是一种通过标准I/O对接外围程序的方法；链入式端口驱动速度更快，但却牺牲了安全性；NIF可以高效接入库函数，同时风险系数也最高。在将JSON解析库集成到Simple Cache的过程中，我们优先采用普通端口。

12.2 用端口来集成解析器

任务已经明确了，我们要把用C写成的YAJL JSON解析器集成到Simple Cache应用。也就是说，要按照上一节中介绍的方法将C库与Erlang进行对接。集成的收益是双方面的：既能用上高效的解析库，又不用大幅修改现有的Erlang代码。集成的工作量固然也不小，但却免除了学习JSON解析的成本，比起用Erlang写一套全新的解析器来说还是省力多了。

在后续章节中，我们还会更进一步，把普通端口改造成链入式端口驱动和NIF。因此在设计库的接口时应尽可能采用模块化的设计。所幸我们的API相当简单：输入JSON文档，执行解析，最后将解析出的Erlang数据结构返回给调用方即可。

正如12.1节所述，在集成外围代码时应优先考虑普通端口，除非理由充分，否则一般不采用链入式端口驱动或NIF。既然采用普通端口，就必须将YAJL库包装成一个以stdin为输入、以stdout为输出的外围程序；YAJL库是用C写成的，用C来编写这个外围程序自然也最为简便高效。在集成过程中，端口创建、消息发送和端口数据处理等功能将由Erlang代码负责完成。这部分代码比较简单，而且可以很好地演示端口通信，我们不妨先从此处入手。

12.2.1 Erlang方面的端口

我们曾在12.1.1节中说过，每个端口都隶属于一个Erlang进程（即端口的属主进程），端口输

出的数据最终都会进入属主进程的信箱。属主进程一旦终止，端口就会关闭。由此可见，端口的管理并不复杂；既然本书的主题是Erlang/OTP，我们不妨采用gen_server来实现端口管理功能。此处我们只节选了部分关键代码，其余内容就不一一罗列了。gen_server已经是你耳熟能详的模块，自行补全缺失部分应该不成问题。

按照本书中其他示例的命名规则，我们给这个模块取名为jp_server（jp即JSON Parser的缩写）。与此相对应，负责管理端口的服务器就是与该模块同名的本地注册进程。为了容纳这个模块，我们再创建一个名为json_parser的应用，为应用补齐jp_app和jp_sup模块后，还要编写一个用于封装jp_server的前端API模块json_parser。

1. 服务器端API

首先从jp_server API函数入手：除去用于启动服务器的start_link/0以外，该模块只有一个API函数，即parse_document/1。如下：

```
parse_document(Data) when is_binary(Data) ->
    gen_server:call(?SERVER, {parse_document, Data}).
```

一如往常，这个API只是对gen_server 中特定函数调用的一个简单包装。本例中的服务器理应对所有请求作出应答，因此我们选用了同步的gen_server:call/2。参数Data是个二进制串，内含待解析的JSON文档。调用parse_document/1将促使gen_server调用handle_call/3回调，回调的第一个调用参数为{parse_document, Data}。Erlang与端口之间的通信便就此正式铺开。

2. 在服务器启动的同时打开端口

在介绍向端口传送数据的方法之前，我们先来看看如何在json_server进程启动的同时打开端口。这样做是为了让端口能在第一时间就绪。这一动作应放入gen_server的init/1回调。除去其他初始化工作，init/1还应调用BIF open_port/2打开端口并将端口标识符存入服务器状态，如下所示：

```
case code:priv_dir(?APPNAME) of
    {error, _} ->
        error_logger:format("~w priv dir not found~n", [?APPNAME]),
        exit(error);
    PrivDir ->
        Port = open_port({spawn, filename:join([PrivDir, "jp_prog"])},
                         [binary, {packet, 4}, exit_status]),
        {ok, State#state{port=Port}}
end
```

（此处的APPNAME宏被定义为应用名，即本例中的json_parser，gen_server的状态记录中新增了字段port。）上述代码首先尝试获取应用中priv子目录的路径，在路径的末尾追加外围程序名jp_prog后，再调用open_port({spawn, ProgramPath}, Options)启动外围C程序。下一节将详细讨论这个C程序。

注意 应用中的外围程序通常都放在priv目录（或其子目录）下。以应用名为参数调用code:priv_dir/1可以查询任意应用的priv目录路径。

应用目录和代码路径

在调用 `priv_dir/1` 等 code 模块中的函数时，它们会在代码路径中搜索给出的应用名。例如，调用 `priv_dir(foo)` 时，如果代码路径中包含.../foo/ebin，则 `priv_dir` 函数会返回对应的目录名.../foo/priv。对于线上系统来说，应用启动之前会事先设置好路径，不会造成什么问题；但在自己的电脑上用 Erlang shell 做测试的时候，我们往往会用 erl -pa ./ebin 来启动 Erlang。在这种情况下，系统并不知道该到何处去查找应用，进而导致 `priv_dir/1` 无法定位 foo 应用。碰到这个问题的时候，只需换用 erl -pa ../foo/ebin 等方式来启动 Erlang 就可以了。

`open_port/2`的第二个参数是一张选项列表，端口将根据这些选项来决定如何处理往返于Erlang和外围代码之间的数据。本例中用到的选项如下。

- `binary`表示端口发送给Erlang的数据应采用二进制串格式而非字节列表格式。
- `{packet, N}`让Erlang在发送给外围代码的每块数据前加上一个*N*字节长的整数前缀（*N*可取值为1、2或4），用于指定后续跟着多少字节。这样做可以在一定程度上简化C代码。
- `exit_status`表示当外围程序退出时，退出状态码应作为数据的一部分发送给Erlang。

更多详情请参见Erlang/OTP文档中的`erlang:open_port/2`。

现在端口已经打开，端口的标识符也已经存入进程状态，下面我们来看看应该如何处理`{parse_document, Msg}`请求。

3. 端口通信

以下代码负责向端口发送消息并等待应答，端口收到消息后会转发给外围代码：

```
handle_call({parse_document, Msg}, _From, #state{port=Port}=State) ->
    Port ! {self(),{command, term_to_binary(Msg)}},
    receive
    {Port, {data, Data}} ->
        {reply, binary_to_term(Data), State}
end.
```

注意到此处用到了`term_to_binary(Msg)`，该调用会按Erlang的标准外部传输格式将`Msg`编码成二进制数据。换言之，可以向外围程序发送任意Erlang数据，在下一节中你将会看到，外围程序可以利用Erl_Interface ei库来解码来自Erlang的二进制数据。（本例中由`parse_document/1` API函数发送的`Msg`本身就是个二进制串。不过没关系，上述代码能够将任意Erlang项式转换成可经由端口传输的外部格式。）

请求处理完毕之后，外围程序会将解析出的数据结构组织成`{Port, {data, Data}}`格式的消息发送给正在等待应答的gen_server服务器，也就是端口的属主进程。外围程序发送的数据也遵循同样的格式，因此只需调用`binary_to_term/1`便可以将之转换为Erlang项式。

你可能已经注意到了，这段代码有些蹊跷。一般来说gen_server的请求处理延迟应该尽可能低；但在处理较大的文档时，这段代码却有可能要阻塞上好一段时间才能从端口处收到应答。不过对于jp_server来说，这倒不是问题：本例中的外围程序一次只能处理一个客户端，因此端口不会面临并发请求——但这样一来，哪个应答该回复给哪个客户端，其间的对应关系就必须由你自

己来维护了。简单起见，我们要求服务器在端口通信完成前不得接受新的请求，这样做实际上是将并发控制交给了gen_server（在集成硬件驱动等程序时这种手法尤为有效）。因此，这个版本的jp_server一次只能解析一份文档，还不具备并发处理能力。

4. 故障检测

由于在创建端口时用上了exit_status选项，外围程序退出时服务器会收到一条带外消息。你可以根据这条消息来跟踪和管理外围程序，譬如通过重新打开端口来重启外围程序，详情见下：

```
handle_info({Port, {exit_status, Status}}, #state{port=Port}=State) ->
    error_logger:format("port exited with status ~p; restarting",
                        [Status]),
    NewPort = create_port(),
    {noreply, State#state{port=NewPort}}.
```

（上述代码假设create_port()函数封装了打开端口的操作细节。）

相较之下，另一种方法可能更为优雅，那就是把上述的gen_server服务器彻底改造成外围程序的代理。我们不妨把服务器设置成监督者管辖范围之下的一个透明（transient）子进程；一旦发现外围程序退出且状态码不为零，就关闭服务器（注意服务器退出时原因码不能是normal）。监督者发现服务器异常终止后，便会自动重启整个服务器，外围程序进而也会随之重启。该方案的优点在于可以复用SASL的日志功能和OTP的重启策略。方案的细节就不在此赘述了，不过我们鼓励你按照这个思路去做做实验。

12.2.2　C方面的端口

上一节中编写的Erlang代码要求应用的priv子目录下放有一个名为jp_prog的可执行程序。现在我们就开始用C语言编写这个程序。不是C语言达人？没关系，大部分代码应该都看得懂。不过要想亲手实验的话，你的机器上必须装有C编译器，比如gcc。至于Windows平台，不妨试试MinGW，这是gcc编译器的一个移植版本，还附带一组可执行简单脚本和makefile的精简UNIX工具套件。

如本章开头所述，这个C程序负责包装YAJL库，是个独立的可执行程序。它从标准输入流中读入JSON文档，调用YAJL进行解析，再通过Erl_Interface ei库将解析结果编码成Erlang项式，最后将项式以字节片段的形式写入标准输出流。整个过程与Erlang端口的要求严丝合缝。

要不要用 Erl_Interface

请注意，用 C 开发外围代码时，Erl_Interface 并不是唯一的选择。因地制宜才是上策——如果在应用中 Erlang 与 C 之间的数据传输只涉及纯文本或成块的字节片段，完全不涉及结构化的 Erlang 数据，那么不妨选用一种简单的面向字节的协议，再辅以自定义的编解码策略。Erl_Interface 的优势在于替你包办了这些事务，而且端口两边都可以使用 Erlang 项式。相较之下这个方案更为重型，但在 C 代码需要发送复杂 Erlang 项式的情况下，选用 Erl_Interface 会更为省心。

要知道，在存放Erlang源码文件的src目录下通常是不混放其他语言的源码文件的。因此，我们将C文件放在另建的c_src目录下。代码清单12-1罗列了c_src/jp_prog.c的第一部分，主要是若干简单的`include`语句和一些必要的声明。

代码清单12-1　c_src/jp_prog.c：声明部分

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ei.h>                                      ❶ Erl_Interface和YAJL的头文件
#include <yajl/yajl_parse.h>

#define BUFSIZE 65536
                                                     ❷ 解析器回调函数声明
static int handle_null(void *ctx);
static int handle_boolean(void *ctx, int boolVal);
static int handle_integer(void *ctx, long integerVal);
static int handle_double(void *ctx, double doubleVal);
static int handle_string(void *ctx, const unsigned char *stringVal,
                         unsigned int stringLen);
static int handle_map_key(void *ctx, const unsigned char *stringVal,
                          unsigned int stringLen);
static int handle_start_map(void *ctx);
static int handle_end_map(void *ctx);
static int handle_start_array(void *ctx);
static int handle_end_array(void *ctx);
                                                     ❸ 让YAJL获知回调
static yajl_callbacks callbacks = {
  handle_null,
  handle_boolean,
  handle_integer,
  handle_double,
  NULL,  /* can be used for custom handling of numbers */
  handle_string,
  handle_start_map,
  handle_map_key,
  handle_end_map,
  handle_start_array,
  handle_end_array
};
                                                     ❹ 用于跟踪嵌套数据结构
typedef struct container_t {
  int index;    /* offset of container header */
  int count;    /* number of elements */
  struct container_t *next;
} container_t;

typedef struct {                                     ❺ 保存YAJL解析过程中的状态
  ei_x_buff x;
  container_t *c;
```

```
  char errmsg[256];
} state_t;

#define ERR_READ          10
#define ERR_READ_HEADER   11      退出状态码
#define ERR_PACKET_SIZE   12
```

要使用Erl_Interface ei库和YAJL库中的函数和数据结构，必须先包含这两个库的头文件❶。然后，声明后文将要实现的回调函数，逐一注明函数名和类型签名，以便后续引用❷。接下来，在yajl_callbacks结构体中填入刚刚声明的回调函数❸，以供YAJL后续调用。其中一个字段被置为NULL：该字段对应的回调负责处理整数与双精度浮点数，可用于处理无法放入长整型的大数。

1. 数据结构

还有一些数据结构是供你自己使用的。首先就是container_t结构体❹。YAJL在处理JSON数组和键/值对映射时需要临时记录一些辅助信息，container_t的作用就是保存这些信息。JSON的这两种数据结构都是嵌套结构，因此必须得找出一种用于表示嵌套容器的方法，这便是container_t中next字段的作用。count字段用于记录当前容器中已经处理完毕的元素数目——比如解析器正在处理某数组的第四个元素，而该数组本身又是另一个键/值对映射的第七个元素。最后一个字段index用于修订最终的数据结构：以JSON数组为例，在解析数组的过程中，只有解析到末尾时才能确定当前数组中到底有多少个元素，这时应折回正在构造的Erlang项式的起始处，将元素总数填入项式头部；index字段中记录的就是最终填写元素总数的位置。详情参见后文。

第二个数据结构是state_t❺，用于保存YAJL解析器在运行期间的全局状态。每个YAJL回调函数的首个参数都是指向该结构体的指针（在YAJL文档中，该参数被称作context）。在Erlang中也有类似的做法，比如gen_server总是将服务器当前的状态作为各个回调函数的最后一个参数。代码中定义的解析器状态包含一个指向当前容器的指针（解析器不处在容器内时指针为NULL）和一个由Erl_Interface ei库定义的ei_x_buff结构体，下文将用该结构体来构造Erlang项式。（运用ei_x_buff时，ei库会自动为你构造的Erlang项式动态分配内存，从而简化编程。）此外，还有一个用于报错的字符串缓冲区。

2. 数据读写

首先登场的是负责处理输入输出的代码。前文曾经做过介绍，Erlang方面的通信是通过标准I/O流来完成的，而且收发的数据包都带有一个按网络字节序（大端序）排列的四字节整数前缀，用以指代后续数据的长度。

> **注意网络字节序**
>
> 无论运行于哪个平台，数据包头部用于指定数据长度的整数前缀都是按网络字节序（大端序）排列的。因此代码必须能够在任意平台上正确解读数据包长度。尤其需要注意的是，千万不能把这几个字节直接当作uint32_t去解读。

之所以是这样一种格式，是因为打开端口时指定了{packet, 4}选项，详情请参见12.2.1节。

采用这一格式，负责解读数据包长度的C代码会略显复杂；但另一方面，读出数据包长度之后剩余数据的读取过程可以大大简化。同理，在向Erlang发送数据时也必须先按正确的格式发送数据包头部长度。发完长度之后，剩余数据只需一次写操作便可发送完毕。代码清单12-2中便是I/O处理相关的代码。

代码清单12-2　c_src/jp_prog.c：数据读写

```
static void write_packet(char *buf, int sz, FILE *fd)      ❶ 输出包含头部长度信息的数据
{
  uint8_t hd[4];
  hd[0] = (sz >> 24) & 0xff;
  hd[1] = (sz >> 16) & 0xff;
  hd[2] = (sz >> 8) & 0xff;
  hd[3] = sz & 0xff;
  fwrite(hd, 1, 4, fd);

  fwrite(buf, 1, sz, fd);
  fflush(fd);
}

static size_t read_bytes(unsigned char *buf, size_t max, FILE *fd)      ❷ 将数据读入缓冲区
{
  size_t n;
  n = fread(buf, 1, max, fd);
  if ((n == 0) && !feof(fd)) {
    exit(ERR_READ);
  }
  return n;
}

static void read_packet(unsigned char *buf, size_t max, FILE *fd)      ❸ 读取数据头部长度
{
  size_t n, sz;
  uint8_t hd[4];

  n = read_bytes(hd, 4, fd);
  if (n == 0 && feof(fd)) exit(EXIT_SUCCESS);
  if (n != 4) exit(ERR_READ_HEADER);
  sz = (hd[0] << 24) + (hd[1] << 16) + (hd[2] << 8) + hd[3];
  if (sz > max) {
    exit(ERR_PACKET_SIZE);
  }
  n = read_bytes(buf, sz, fd);
  if (n != sz) {
    exit(ERR_READ);
  }
}
```

write_packet函数❶用于输出完整的数据包，其中每个数据包附带一个四字节长的网络字节序整数前缀，用于指定后续数据的长度。工具函数read_bytes❷用于读取数据并将之填入缓冲区（最多读取max个字节），read_packet函数❸用于读取带有长度前缀的数据包。请注意，读写操作一旦失败，程序就会以非零状态码退出。在通信故障或协议不同步的情况下，往往难以

准确区分数据包头和原始数据，因此最好还是直接退出，让Erlang重启程序为妙。只有当输入流恰好在JSON文档读取完毕时关闭，程序才会正常退出。

3. 运行JSON解析器

接收了Erlang方面传入的文档并将之存入缓冲区之后，就可以初始化YAJL解析器并开始解析收到的数据了。该任务由parse_json函数负责（参见代码清单12-3）。请注意，该函数的第一个参数是个指向state_t结构体的指针；收到该指针后，YAJL会将它传递给你定义的各个回调函数，以便它们随时访问解析器的当前状态。

代码清单12-3　c_src/jp_prog.c：JSON解析

```
static const char *parse_json(state_t *st, unsigned char *buf, size_t len)
{
  yajl_parser_config cfg = {
    1, /* allow comments */
    0  /* don't check UTF-8 */
  };
  yajl_handle yh;
  yajl_status ys;
  const char *err=NULL;
                                                          ❶ 初始化YAJL解析器句柄
  yh = yajl_alloc(&callbacks, &cfg, NULL, st);
  ys = yajl_parse(yh, buf, len);
  if (ys == yajl_status_insufficient_data) {
    ys = yajl_parse_complete(yh);
  }
  if (ys == yajl_status_insufficient_data) {
    err = "unexpected end of document";
  } else if (ys != yajl_status_ok) {
    unsigned char *msg = yajl_get_error(yh, 0, NULL, 0);
    strncpy(st->errmsg, (char *)msg, sizeof(st->errmsg)-1);
    yajl_free_error(yh, msg);
    st->errmsg[sizeof(st->errmsg)] = 0;
    err = st->errmsg;
  }
                                                       ❷ 释放YAJL句柄
  yajl_free(yh);
  return err;
}
```

该函数首先在开头处为YAJL创建了几样东西：一个配置信息结构体、一个解析器句柄和一个用于表示当前解析情况的变量。接着，调用yajl_alloc初始化句柄❶，调用参数包括代码清单12-1中定义的那套回调函数、配置信息结构体，以及状态指针（context变量）。第三个参数暂时设为NULL。句柄初始化完毕后便可以利用该句柄来调用yajl_parse去解析数据缓冲区中的数据了。在解析过程中，YAJL会伺机调用你提供的回调函数来构建所需的结果。解析完成后，需要调用yajl_free释放句柄❷。

如果yajl_parse的返回值指示文档不完整，则应调用yajl_parse_complete告知解析器数据已经全部处理完毕；如果仍旧显示不完整，则令函数返回，并在错误信息中注明文档的确不完整。如果解析过程出现其他异常，则调用yajl_get_error获取错误字符串并将之复制到解析

状态结构体的字符串缓冲区内，然后用yajl_free_error释放句柄。为了简化parse_json的使用，不妨对函数的返回值稍作修饰：解析成功返回NULL，解析失败则返回内含错误信息的字符串。

4. 用ei编解码Erlang项式

搞定这些之后，现在我们来研究一下单个请求包的处理过程。jp_prog收到Erlang发来的请求包后会将之存入输入缓冲区。剩下的工作便交由process_data函数完成，详情参见代码清单12-4，其中还包括一个辅助函数make_error。截至目前为止，我们只讨论了数据读写和YAJL解析器的调用方法；接下来，我们将阐述如何在C代码中调用ei库处理Erlang项式。

代码清单12-4　c_src/jp_prog.c：解析单个文档

```
static void make_error(state_t *st, const char *text)
{
  ei_x_free(&st->x);
  ei_x_new_with_version(&st->x);
  ei_x_encode_tuple_header(&st->x, 2);
  ei_x_encode_atom(&st->x, "error");
  ei_x_encode_string(&st->x, text);
}

static void process_data(unsigned char *buf)
{
  state_t st;
  st.c = NULL;
  ei_x_new_with_version(&st.x);                    ❶ 初始化输出缓冲区

  int index = 0;
  int ver = 0, type = 0, size = 0;
                                                       ❷ 解析输入数据中的版本字节
  if (ei_decode_version((char *)buf, &index, &ver)) {
    make_error(&st, "data encoding version mismatch");
  } else if (ei_get_type((char *)buf, &index, &type, &size)
             || type != ERL_BINARY_EXT) {
    make_error(&st, "data must be a binary");
  } else {                                             ❸ 对{ok, ...}的开头进行编码
    ei_x_encode_tuple_header(&st.x, 2);
    ei_x_encode_atom(&st.x, "ok");
    const char *err;
    if ((err = parse_json(&st, &buf[index+5], size)) != NULL) {
      make_error(&st, err);
    }
  }
  write_packet(st.x.buff, st.x.buffsz, stdout);
  ei_x_free(&st.x);
}
```

在对Erlang项式进行序列化时，erlang:term_to_binary/1采用的正是Erlang分布式协议中的外部项式格式。有关该格式的详情请参阅标准Erlang/OTP文档中的ERTS用户指南。好在使用这一格式并不需要关注字节层面的详细表示方法——项式的编解码工作全权交由ei库负责即可。在本例中，Erlang编码完毕后发送给端口的项式已经存入输入缓冲区（参见12.2.1节）。该项

式是个内含JSON文本的二进制串。

在对项式进行解码并开始解析之前，首先要初始化解析器状态，尤其是其中的ei_x_buff结构体❶，该结构体将用于构造最终发回Erlang的解析结果。初始化工作由ei_x_new_with_version完成，具体来说就是分配一个动态缓冲区，再在缓冲区的起始处插入一个版本号。Erlang外部项式格式要求数据块在第一个字节处注明编码方式的版本号——其作用在于确保各节点和客户端能够以相互兼容的方式进行数据交换。一般来说，版本号到底是多少并不重要；ei库中的函数会帮你打理好一切。

ei库中的解码函数用到了一个缓冲区下标变量，它的初值为零，并且会随着解码操作的推进逐步递增。在解码输入数据时，首先应调用ei_decode_version❷确认版本号。如果输入缓冲区中的版本号与ei不兼容，ei_decode_version将返回一个非零值，这时process_data会向Erlang发回一个包含错误信息的字符串项式。如果解码成功，解析出的版本号将被存入变量ver（版本号本身在此处没有意义）。

下一步，我们开始分析藏在版本号字节之后真正待解码的项式。具体方法与上述过程类似：首先调用ei_get_type，若返回值非零则数据无效，否则目标项式的类型和大小将被分别存入变量type和size（该函数不会递增index）。不出意外的话，得出的类型应该是ERL_BINARY_EXT，表示这个项式是个二进制串。此时输入缓冲区中的内容如图12-3所示。

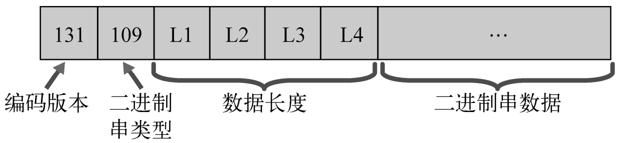


图12-3　以外部项式格式存入输入缓冲区中的二进制串项式。该格式的具体细节并不重要，ei库会协助你完成解码工作

现在，待解码的二进制串就位于buf[index]处。为了避免数据复制的开销，不妨作个弊：根据外部项式格式文档中的描述，在二进制串的编码格式中，类型（109）占1字节，数据长度占4字节（已经解码完毕并存入变量size），实际的二进制数据就位于buf[index+5]处。因此，将这个地址连同数据长度一起直接交给parse_json即可。

在开始解析之前，还得考虑一下应该向Erlang返回什么样的数据。为了方便Erlang识别解析过程的成败，我们决定采用惯用的{ok, Result}/{error, Message}格式。这样一来，在对解析结果数据进行编码之前，必须先对元组{ok, ...}进行编码❸。

编码过程与解码过程相反：现有的动态ei_x_buff输出缓冲区会自动记录缓冲区的当前大小和偏移，后续数据会从当前偏移位置写入。构造元组时应首先调用ei_x_encode_tuple_header向缓冲区插入一个元组头部，表示后面跟着一个特定大小的元组。（元组的元素本身也可以是元组，从而形成嵌套结构。）本例中我们要写入的是一个二元组，其中第一个元素是原子ok，是通过调用ei_x_encode_atom插入，第二个元素将由你定义的YAJL回调在解析过程中构造而成。

工具函数make_error用于报错。现在ei_x_buff输出缓冲区已经初始化完毕，而且有可能已经插入了部分数据，因此不能直接在make_error中使用，必须先将它释放再另行初始化新的缓

冲区。接着便可以开始构造{error, Message}元组了，具体方法与构造{ok,...}元组时大同小异，只是在构造包含错误信息的Message项式时应该调用ei_x_encode_string。

最后，write_packet函数负责将输出缓冲区中编码完毕的项式（可能是{ok, Data}，也可能是{error, Message}）写入标准输出流，并释放输出缓冲区。

5. 主函数

开始研究YAJL回调函数之前，还有一个细节有待考察，那就是程序的主循环，如代码清单12-5所示：

代码清单12-5　c_src/jp_prog.c：主循环

```
int main(int argc, char **argv)
{
  static unsigned char buf[BUFSIZE];
  for (;;) {
    read_packet(buf, sizeof(buf)-1, stdin);
    buf[sizeof(buf)-1] = 0; /* zero-terminate the read data */
    process_data(buf);
  }
}
```

main函数是所有C程序的入口点。它的两个参数分别是命令行参数的个数（argc）和实际传入的命令行参数字符串（argv），不过这个程序还用不到命令行参数。由于实质工作都被转入了其他函数，main函数本身十分简洁：首先为输入的数据创建缓冲区，然后进入无穷循环（直至代码清单12-2中的某个函数调用exit为止），每循环一次读取并处理一个发自Erlang的数据包。

6. 将JSON数据解析成Erlang项式

现在开始讲解这个C程序的最后一部分代码：即YAJL解析器回调函数。YAJL在解析JSON数据时会伺机调用这些回调。null、true/false、数值、字符串等简单类型各有一个回调。复合JSON数据结构（数组和映射）则在头尾两处分别对应于一个回调。

我们曾经说过，每个回调的参数列表中都包含一个指向state_t结构体的指针（即YAJL的context，参见代码清单12-3）。构建对应的Erlang项式时，一切相关信息都记录在该结构体内。在C中，面对这类任务时我们往往会经不住使用全局变量的诱惑；但真要是这么做了的话，等到12.3节换用链入式驱动时免不了还是要推翻重写，因为链入式驱动要求代码必须可重入。

在编写回调函数之前，必须先明确应该用什么样的Erlang项式来表示各种JSON数据。二者之间的对应关系参见表12-1。

表12-1　用Erlang项式表示JSON数据。其中`null`被转换成了更符合Erlang规范的原子`'undefined'`

JSON	Erlang中的`json()`类型
`null`	`'undefined'`
`true`	`'true'`
`false`	`'false'`
`42, 3.14, ... (integers and floats)`	`number()`
`"..." (string)`	`binary()`
`[x1, x2, ...]  (array)`	`{ json(), json(), ... }`
`{"abc": x1, "def": x2, ...}  (map)`	`[ {binary(), json()}, ... ]`

我们选用的表示法首先应该尽量遵循Erlang规范（比如用原子'undefined'来表示null），其次要省空间（比如用二进制串来表示JSON字符串和映射中的标签），最后还不能有歧义。看到列表，就知道它表示的肯定是键/值对映射，不会是字符串或数组。看到不是键/值对的元组，就知道它表示的肯定是数组。用Erlang元组来表示JSON数组更便于用下标访问数组，而且也比列表更省空间；但另一方面，增删数组元素时不得不先临时将元组转换成列表。权衡之后我们认为这个代价是值得的。

现在我们已经知道输出结果应该长成什么样子了，下面先来看几个简单的YAJL回调，如代码清单12-6所示。

代码清单12-6　c_src/jp_prog.c：简单YAJL回调

```
static void count_element(state_t *st)
{
  container_t *c = st->c;
  if (c != NULL) ++(c->count);          ❶ 递增当前容器中的元素数目
}

static int handle_null(void *ctx)
{
  state_t *st = (state_t *)ctx;
  count_element(st);
  ei_x_encode_atom(&st->x, "undefined");
  return 1;
}

static int handle_boolean(void *ctx, int boolVal)
{
  state_t *st = (state_t *)ctx;
  count_element(st);
  ei_x_encode_boolean(&st->x, boolVal);
  return 1;
}

static int handle_integer(void *ctx, long integerVal)
{
  state_t *st = (state_t *)ctx;
  count_element(st);
  ei_x_encode_long(&st->x, integerVal);
  return 1;
}

static int handle_double(void *ctx, double doubleVal)
{
  state_t *st = (state_t *)ctx;
  count_element(st);
  ei_x_encode_double(&st->x, doubleVal);
  return 1;
}

static int handle_string(void *ctx, const unsigned char *stringVal,
                         unsigned int stringLen)
{
```

```
  state_t *st = (state_t *)ctx;
  count_element(st);
  ei_x_encode_binary(&st->x, stringVal, stringLen);   <-- ❷ 将JSON字符串编码为二进制串
  return 1;
}

static int handle_map_key(void *ctx, const unsigned char *stringVal,
                          unsigned int stringLen)
{
  state_t *st = (state_t *)ctx;                        ❸ 开始编码用于表示键/值对的二元组
  ei_x_encode_tuple_header(&st->x, 2);               <--
  ei_x_encode_binary(&st->x, stringVal, stringLen);
  return 1;
}
```

上面所有函数都用到了辅助函数`count_element`，该函数用于递增当前容器结构体（如果存在的话）中的元素计数（参见代码清单12-1）❶。例如，如果在某数组中发现一个整数，那么该数组的元素计数便会加一。了解这一点后，这些函数的作用就显而易见了，无非就是向输出缓冲区中插入各种数据罢了。以`handle_string`为例，该回调的作用就是调用`ei_x_encode_binary`将JSON字符串转换成缓冲区中的二进制串❷。这些回调的返回值都是1，表示"一切正常，请继续解析"。其中有一个函数特别值得注意，它就是`handle_map_key`。该函数用于解析JSON映射键/值对中的键。

在解析过程中碰到键/值对时，解析器一定处在映射的解析过程当中，因此必然已经准备好了用于存放键/值对的列表。每个键/值对都是个由键和值构成的二元组；和代码清单12-4中的`{ok, ...}`元组的编码方法一样，只需插入一个二元组的头部❸，再在第一个元素的位置上插入（二进制串格式的）键即可。下一个回调函数会在第二个元素的位置上插入与键相对应的值，从而将二元组补全。很简单吧？（请注意，只有在处理键/值对中的值时才能调用`count_element`，处理键时不能调用，因为只有在整个元组处理完毕之后才能递增映射的元素计数。）

YAJL 怎样处理键/值对

在各个 YAJL 回调函数中，只有键/值对中的键所对应的回调函数比较特殊。值的部分无须特殊处理——直接调用普通的回调函数即可。如有必要，可以在状态结构体中添加一个标志位，用于标识下一个值是否隶属于某个键/值对，我们将在 12.3 节中运用这一手法，不过目前还没有做此特殊处理的必要。

最后就轮到数组和映射了。这两种数据结构在头尾两处必须区别对待，因而一共用到了4个回调函数。这部分代码有点儿复杂，参见代码清单12-7。

12

代码清单12-7　c_src/jp_prog.c：映射和数组的YAJL回调

```
static int handle_start(void *ctx, int array)
{
  state_t *st = (state_t *)ctx;
  count_element(st);
```

```
  container_t *c = malloc(sizeof(container_t));
  c->next = st->c;
  st->c = c;
  c->count = 0;
  c->index = st->x.index;

  if (array) {
    ei_x_encode_tuple_header(&st->x, 1);
  } else {
    ei_x_encode_list_header(&st->x, 1);
  }
  return 1;
}

static int handle_start_map(void *ctx)
{
  return handle_start(ctx, 0);
}

static int handle_start_array(void *ctx)
{
  return handle_start(ctx, 1);
}

static int handle_end(void *ctx, int array)
{
  state_t *st = (state_t *)ctx;
  container_t *c = st->c;
  if (array) {
    ei_encode_tuple_header(st->x.buff, &c->index, c->count);
  } else {
    ei_encode_list_header(st->x.buff, &c->index, c->count);
    ei_x_encode_empty_list(&st->x);
  }
  st->c = c->next;
  free(c);
  return 1;
}

static int handle_end_map(void *ctx)
{
  return handle_end(ctx, 0);
}

static int handle_end_array(void *ctx)
{
  return handle_end(ctx, 1);
}
```

❶ 分配新容器并进行链接和初始化

❷ 插入元组或列表临时头部

❸ 将最终的元素数填入头部

❹ 摘除并释放容器结构体

本例中映射和数组的处理方式极为相似。简便起见，我们将相似的代码合并成了handle_start和handle_end两个公共函数，并用一个标志变量来表示当前处理的是数组还是映射。

handle_start首先递增当前容器的元素计数——显然，位于数组中的数组应该算作其父容器的元素，而不能算作它自身的元素。接下来，用malloc分配一个用于指代当前容器的新的容器

结构体，然后把它存入解析器状态并清零元素计数。另外请注意，务必要记下ei_x_buff的当前下标，待会儿我们还要回到这里做一些收尾工作❶。一切就绪之后，根据正在构建的是数组还是映射，再相应插入一个元组头部或列表头部❷。由于暂时还不知道容器中确切的元素数目，头部中的元素计数暂设为1。

当handle_end被调用时，容器中的确切元素数目已经揭晓。这时就可以更新先前写入容器头部的元素数目了。这个更新操作可通过调用ei_encode_tuple_header来完成（对于列表应调用ei_encode_list_header）❸，调用参数分别是指向当前ei_x_buff缓冲区起始位置的指针、先前保存在容器结构体中的缓冲区下标以及容器中的元素数目。ei_encode_系列函数与ei_x_encode_系列函数的区别在于前者将内存管理全权交由开发者负责，因此它们不会修改ei_x_buff结构体中的当前下标——它只会覆盖先前插入缓冲区的临时容器头部。如果处理的是列表，还需要调用ei_x_encode_empty_list插入一个空表——注意这个空表位于列表中最后一个元素的后面，而不是在表头的后面（因为调用的是ei_x_encode_...函数——译者注）。至此，输出缓冲区的状态如图12-4所示。

最后，将容器结构体从解析器状态中摘除并释放内存❹。

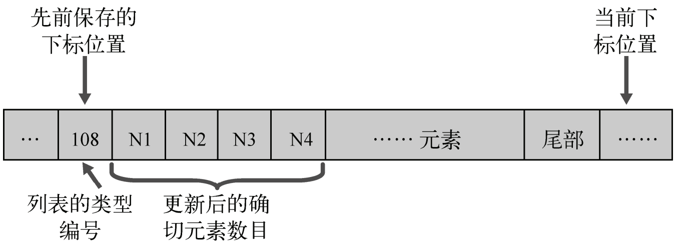


图12-4 在列表头部补上最终的长度。先前保存的下标指向列表头部的起始处，而动态的ei_x_buff的当前下标则指向列表尾部之后的第一个字节

呼！可真是够辛苦的。不过这个示例还没结束，还有两种集成方法有待介绍。在下一节中，我们将引入链入式驱动，好在所需修改的代码并不多。先前为优化代码结构而付出的努力总算是有了回报。至于现在，不妨先把代码跑起来看看吧。

12.2.3 编译运行

请将src下的.erl文件逐一编译为ebin目录下的.beam文件。c_src/jp_prog.c则应编译为priv目录下的jp_prog可执行文件，以供Erlang代码通过端口调用。此外，还需要编写一个名为json_parser的前端API模块，该模块应导出以下函数：


```
parse_document(Data) ->
    jp_server:parse_document(Data).
```

请用以下命令行调用gcc编译器编译C代码：

```
$ gcc -o ./priv/jp_prog -I${OTPROOT}/lib/erl_interface-3.6.5/include
➥ -I${YAJLROOT}/include -L${OTPROOT}/lib/erl_interface-3.6.5/lib
➥ -L${YAJLROOT}/lib ./c_src/jp_prog.c -lei_st -lyajl
```

在执行上述命令之前，必须先定义两个环境变量，分别是指向Erlang安装路径的OTPROOT（一般是/usr/lib/erlang，如果弄不清楚具体路径，请在Erlang中调用code:root_dir()）和指向YAJL安装路径的YAJLROOT。（如果下载YAJL之后只编译不安装，则该路径应类似于lloyd-yajl-1.0.9-0/build/yajl-1.0.9/。）请注意命令行内必须用-I选项指定Erl_Interface和YAJL的头文件路径，用-L选项指定链接过程中库文件的查找路径，以及用-l选项指定要链接的库的名称。另外还需要注意命令行中源码文件和库文件出现的次序。

现在priv目录下的jp_prog可执行文件应该已经准备就绪。由于所有输入都带有四字节长的前缀，直接在命令行下进行测试比较困难——在Erlang中测试反倒更容易些。不过在启动jp_prog之前，必须先告知系统YAJL库的位置。如果YAJL安装在非标准位置下，请按下列方式修改LD_LIBRARY_PATH环境变量：

```
$ export LD_LIBRARY_PATH=${YAJLROOT}/lib
```

如果你用的是Mac OS X，则应修改DYLD_LIBRARY_PATH：

```
$ export DYLD_LIBRARY_PATH=${YAJLROOT}/lib
```

然后，在Erlang中启动json_parser应用，并尝试解析若干JSON文档。（记得在-pa选项中加上json_parser目录，否则code:priv_dir/1无法找到应用目录，参见12.2.1节。）

```
$ erl -pa ../json_parser/ebin
...
1> application:start(json_parser).
ok
2> Doc = <<"[null, true, {\"int\": 42, \"float\": 3.14}]">>.
...
3> json_parser:parse_document(Doc).
{ok,{undefined,true,[{<<"int">>,42},{<<"float">>,3.14}]}}
4>
```

成功了！可以看到null被转换成了undefined，映射中的键从字符串变成了二进制串，最外层的JSON数组被转成了元组，而映射也被表示成了键/值二元组的列表。

C库的集成工作终于完成了，一切都严丝合缝。然而在生产环境下，当服务器负载较高时，序列化/反序列化以及管道通信的开销会变得难以忍受。好在我们已经有了应对之策——将程序改造成链入式驱动。

12.3　开发链入式驱动

整合外围代码时，一般不会一上来就用链入式驱动。不过现在呢，负责对接外围程序的端口已经稳定可用，只是性能方面有些拖后腿。在这种情况下，就可以考虑换用链入式驱动了。

之前用C编写的解析器运转正常，没有必要再做修改。只需对解析器的逻辑加以包装，让Erlang能够加载并启动解析器，进而与之建立通信就可以了。说白了就是把main函数换成符合erl_driver API要求的回调函数，再补充若干链入式驱动专用的辅助数据结构及函数。不过现在，我们得先看看链入式驱动的代码与外围程序的代码都有哪些不同。

12.3.1 初识链入式驱动

我们曾经在12.1.2节中说过，虽然表面上看链入式驱动和普通端口差不多，但它们的底层机制却大相径庭。普通外围端口程序通过标准I/O与Erlang通信，链入式驱动则通过用户自定义的回调函数来接收数据。回调函数要先处理数据，然后将处理结果返回Erlang。和普通端口背后的外围程序一样，同一段链入式驱动代码也有可能被生存期相互重叠的多个独立端口并发调用。由于在运行时所有这些端口共享Erlang VM的内存，而且不同端口还有可能分属不同的线程，因此驱动代码必须*可重入*（可同时被多个调用方调用），不得依赖任何全局变量或锁。

C程序或库中的长效数据分为两类。第一类是全局变量，也就是C语言中的*外部变量*。这些变量定义在所有函数之外，只要程序还在运行、库还被加载着，它们便一直有效。然而问题在于全局变量只有一份，如果多方并发调用同一段代码，调用方之间便会相互覆盖对方的数据，从而导致数据损坏乃至程序崩溃。

举个例子，假设你写了一个提供计数累加功能和当前计数查询功能的简单端口驱动。在不考虑可重入性的情况下，完全可以用C的全局变量来实现计数器。如果一次只为一个调用方服务，完全没有问题。现在假设用户需要5个计数器，并为此启动了5个运行着同一套驱动代码的端口。不幸的是，由于5个端口累加的是同一个全局变量，彼此之间的干扰将导致最终结果一团乱麻。在这种情况下，驱动内部状态被多个驱动实例所共享，如图12-5所示。

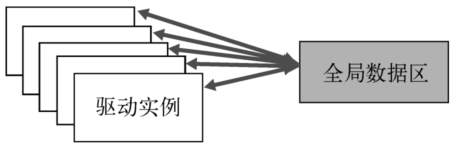


图12-5 不可重入的、使用全局共享内存的链入式驱动代码。所有驱动实例采用同一块内存区域存储自身的数据，这样做很容易导致崩溃

另一类长效数据就是动态分配的内存数据。如果库的调用者自行分配所需的内存，不再干扰其他调用者的内存数据，并发调用就不足为惧了。这正是链入式驱动所要达到的效果。我们将这块内存及其内容称作实例专有数据。仍然沿用上面的计数器驱动示例，如果每个驱动实例都在自己独占的数据区域内维护计数器变量，便万事大吉了：每打开一个端口，都会分配一个单独维护的计数器，各计数器之间互不干扰。如图12-6所示。

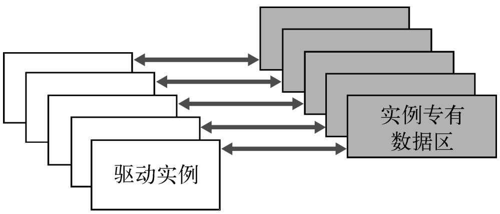


图12-6 仅使用实例专有数据的、可重入的链入式驱动。每个实例都有自己的工作内存，互不干扰

这部分内存可以在驱动初始化时分配，也可以按需动态分配。Erlang erl_driver API中的driver_alloc()和driver_free()等辅助函数就是专为链入式驱动代码的内存管理而设计的。

讨论完这些，就该着手实现了。为外围程序编写的代码大部分都可以在链入式驱动中复用。唯一需要修改的就是Erlang与C代码的对接方式。

12.3.2　驱动的C语言部分

此前的C代码是个带有main函数的独立程序，该程序通过与端口对接的标准输入/输出与Erlang VM通信。在链入式驱动中，由于C代码与Erlang运行时系统运行在同一地址空间内，Erlang VM可以直接调用驱动代码中的回调函数，二者之间的通信便由这组回调函数完成。相关逻辑将会用到erl_driver API，详情请参见Erlang/OTP文档中的ERTS参考手册。

这一节将详细讨论Erlang VM如何与C代码对接。我们将以上一节中的c_src/jp_prog.c为蓝本进行修改，请将该文件复制一份，命名为c_src/jp_driver.c。首先是所有链入式驱动代码都应包含的erl_driver.h头文件：

```
#include <erl_driver.h>
```

另外，由于链入式驱动不使用标准I/O和退出状态码，BUFSIZE、ERR_等宏定义就没有存在的必要了。与此同时，write_packet、read_bytes、read_packet以及main函数，也全都可以删除。

接下来，还要添加一些和erl_driver API相关的定义。如代码清单12-8所示。

代码清单12-8　c_src/jp_driver.c：erl_driver相关定义

```
static ErlDrvData drv_start(ErlDrvPort port, char *command);
static void drv_stop(ErlDrvData handle);
static void drv_output(ErlDrvData handle, char *buf, int sz);

static ErlDrvEntry jp_driver_entry = {                      <-┐
    NULL,                          /* init */                ❶ 驱动结构体
    drv_start,                     /* start */
    drv_stop,                      /* stop */
    drv_output,                    /* output */
    NULL,                          /* ready_input */
    NULL,                          /* ready_output */
    "jp_driver",                   /* driver_name */
    NULL,                          /* finish */
    NULL,                          /* handle (reserved) */
    NULL,                          /* control */
    NULL,                          /* timeout */
    NULL,                          /* outputv */
    NULL,                          /* ready_async */
    NULL,                          /* flush */
    NULL,                          /* call */
    NULL,                          /* event */
    ERL_DRV_EXTENDED_MARKER,       /* ERL_DRV_EXTENDED_MARKER */
```

```
    ERL_DRV_EXTENDED_MAJOR_VERSION, /* ERL_DRV_EXTENDED_MAJOR_VERSION */
    ERL_DRV_EXTENDED_MAJOR_VERSION, /* ERL_DRV_EXTENDED_MINOR_VERSION */
    ERL_DRV_FLAG_USE_PORT_LOCKING   /* ERL_DRV_FLAGs */
};
DRIVER_INIT(jp_driver)                          <-- ❷ 将驱动告知Erlang
{
    return &jp_driver_entry;
}

typedef struct {                                <-- ❸ 实例专用数据的结构体
  ErlDrvPort port;
} drv_data_t;
```

首先声明后续将要实现的各个回调函数。然后，将这些回调告知给Erlang VM。创建一个ErlDrvEntry结构体，在我们打算实现的回调函数的位置上填入相应的函数指针，对于其他回调则填入NULL❶。这个结构体是整个驱动代码的关键所在。其中大部分字段都是各式各样的回调函数指针，通过实现这些回调函数便可完成所需的各种功能。在本例中，我们只关注start、stop和output3个回调。此外，还有一个必须填写的关键字段：driver_name。在打开端口的时候，Erlang将根据这个字段来定位相应的ErlDrvEntry结构体。详情请参见12.3.4节。

DRIVER_INIT宏用于向Erlang VM注册ErlDrvEntry结构体。宏中的驱动名称必须与结构体中的保持一致，但要去掉引号。紧跟宏之后的大括号内会返回一个指向待注册结构体的指针❷。

最后，定义用来存放实例专有驱动数据的结构体，参见图12-6。驱动中的各种需要跨回调调用的、需要长期保存的信息都记录在该结构体内，其作用与gen_server的状态记录类似❸。本例中只需记录Erlang端口，以便告知驱动代码在完成解析后向何处输出结果。我们一般选择在驱动的start回调中为结构体分配内存，该回调将返回一个被转换为特殊指针类型ErlDrvData的指向该结构体的指针。Erlang VM将以调用参数的形式将这个ErlDrvData指针传递给其他驱动回调函数，以供后续随时访问。

1. 端口驱动回调

部分驱动回调函数与驱动的生存期有关，如分别在端口开闭时调用的start和stop。其余回调则在Erlang方面有数据就绪时生效。开发驱动时，可供选择的回调有十几个之多，不过好在并不是每个都要实现一遍。对于大部分驱动而言，必不可少的只有start和stop函数，多半还会带上init。至于通信相关的回调函数，完全就是因地制宜各取所需了。

本例中需要实现的通信回调函数只有output。请注意，除start函数以外，所有回调函数的第一个参数都是由start返回的ErlDrvData句柄。

表12-2列出了可供选用的所有回调。如你所见，选择余地非常之大，不过在单一驱动中不太可能会用上所有回调。

表12-2 可供选用的**erl_driver**回调函数。驱动按需实现，剩下的置为NULL即可

回　调	描　述
init	对于静态链接的驱动，在系统启动时调用；对于动态加载的驱动，在加载完成时调用
start	在执行open_port/1时调用

（续）

回　　调	描　　述
stop	在关闭端口时调用
output	在Erlang进程预备好向端口输出数据时调用。如果启用了outputv回调，则该回调失效
ready_input	当驱动句柄上的输入数据就绪时调用。用于异步I/O
ready_output	当预备好向某端口句柄输出数据时调用。用于异步I/O
finish	在驱动被卸载前调用。仅用于动态加载的驱动
control	类似于用于驱动的ioctl。在Erlang方面调用port_control/3时调用
timeout	在调用驱动定时器时调用
outputv	当有人在Erlang中向端口写入数据时调用。如果未定义该回调，则改用output回调来处理数据写入
ready_async	在driver_async异步调用完成时调用
flush	关闭端口前必须先刷新驱动队列中的数据才能调用stop，flush便在此时调用
call	类似于control回调，但输入输出采用的是Erlang的外部项式格式
event	在driver_event()指定的事件发生时调用

2. 内存管理

时过境迁，现在，驱动代码运行在Erlang VM的地址空间内：可不能再用C的标准库函数malloc和free来管理内存了。开发链入式驱动时，应该换用erl_driver库中的driver_alloc()和driver_free()函数。所幸YAJL库比较灵活，允许你自行制定内存分配函数。只需简单包装一下driver_alloc()、driver_realloc()和driver_free()，再将包装函数的指针填入结构体，让它们能为YAJL所用就可以了。详情参见代码清单12-9。

代码清单12-9　c_src/jp_driver.c：YAJL内存管理回调

```
static void *alloc_func(void *ctx, unsigned int sz)
{
  return driver_alloc(sz);
}

static void *realloc_func(void *ctx, void *ptr, unsigned int sz)
{
  return driver_realloc(ptr, sz);
}

static void free_func(void *ctx, void *ptr)
{
  driver_free(ptr);
}

static yajl_alloc_funcs alloc_funcs = {
  alloc_func,
  realloc_func,
  free_func,
  NULL
};
```

结构体中的最后一个元素是YAJL的上下文指针，这几个函数都以它为第一个参数。（不要把它跟传给其他YAJL回调的上下文指针搞混；除非特别指定，否则二者不一定相同。）不过这里不需要什么上下文，将结构体中的最后一个字段置为NULL即可。最后，让YAJL改用新的内存分配函数就行了。回想一下，parse_json函数是这样调用yajl_alloc的：

```
yh = yajl_alloc(&callbacks, &cfg, NULL, st);
```

设置为NULL的参数就是用于指定yajl_alloc_funcs结构体的。现在，将该调用修改如下：

```
yh = yajl_alloc(&callbacks, &cfg, &alloc_funcs, st);
```

这样就可以让YAJL用driver_alloc来代替malloc了。

代码中还有两处和内存管理相关的位置需要修改。handle_start回调函数中的

```
container_t *c = malloc(sizeof(container_t));
```

应修改为

```
container_t *c = driver_alloc(sizeof(container_t));
```

另外，handle_end函数中的

```
free(c);
```

现在应该改成

```
driver_free(c);
```

用driver_alloc替换malloc使得我们可以用Erlang VM中特殊设计的内存管理函数以线程安全且可重入的方式来接管内存管理。

3. 将数据发回Erlang

当端口有数据需要输出给端口所属的进程时，链入式驱动应调用erl_driver中的driver_output API函数来向Erlang节点回传数据。该调用会将指定缓冲区中的数据写入由第一个调用参数指定的端口。为了访问（记录在实例专有驱动数据结构体中的）端口，应在process_data函数的参数列表中加入一个指向该数据结构体的指针。因此该函数原本的函数原型

```
static void process_data(unsigned char *buf)
```

应修改为

```
static void process_data(drv_data_t *d, unsigned char *buf)
```

然后，在函数体结尾，原有的调用

```
write_packet(st.x.buff, st.x.buffsz, stdout);
```

应该修改为

```
driver_output(d->port, st.x.buff, st.x.buffsz);
```

至此，将原有代码改写为链入式驱动的工作几近完工。剩下的就是驱动回调函数的实现了。

4. 驱动回调实现

最后，我们来看看驱动中3个回调函数的实现：start、stop和output，如代码清单12-10

所示。

代码清单12-10　c_src/jp_driver.c：erl_driver回调

```
static ErlDrvData drv_start(ErlDrvPort port, char *command)
{
  drv_data_t *d = (drv_data_t *)driver_alloc(sizeof(drv_data_t));
  d->port = port;
  return (ErlDrvData)d;
}

static void drv_stop(ErlDrvData handle) {
  driver_free((char *)handle);
}

static void drv_output(ErlDrvData handle, char *buf, int sz)
{
  process_data((drv_data_t *)handle, (unsigned char *)buf);
}
```

首先是在驱动实例启动时（也就是用该驱动打开端口时）调用的drv_start。该函数会创建供其他函数使用的实例专有数据结构体并在其中存入Erlang端口的标识符。随后，指向该结构体的指针将被转换为ErlDrvData类型并被当作调用结果返回。

接下来是在端口关闭时调用的drv_stop，它负责调用driver_free释放实例专有数据结构体。在更为复杂的链入式驱动中，start回调有可能还会启动线程、分配初始化资源等。负责释放这些资源、清理这些线程的，通常就是stop回调。

drv_output函数是本例中用到的唯一一个和通信相关的回调。其功能和12.2节中的外围程序的main函数相近，只是驱动程序与Erlang VM之间的通信方式变了。有别于之前采用的独立操作系统进程之间的字节流，链入式驱动与Erlang之间的通信是由Erlang VM直接调用drv_output回调函数来实现的，序列化和反序列化操作也和VM自身共同运行在同一地址空间内。处理器循环也没有必要了，只要有数据发送至端口处，Erlang VM随时都可以调用驱动程序，只需要将数据指针连同指向实例专有数据的指针一并传递给process_data函数便可。

将新代码与上一节中的程序做个比较，除通信部分以外，其余代码完全一样。这样一来，你便可以因地制宜地作出选择：外围程序安全有余而效率不足；链入式驱动速度更快却有可能导致整个VM崩溃。无论采用的是哪种方式，只要代码结构合理，将代码改写为另一种方式都不困难。

12.3.3　编译驱动代码

我们需要将jp_driver.c文件编译成一个共享库（在UNIX下叫共享目标文件，在Windows下则叫动态链接库）。因此gcc的命令行参数会略有不同。请在命令行中执行以下命令：

```
gcc -o ./priv/jp_driver.so -fpic -shared -I${OTPROOT}/erts-5.7.5/include
➥ -I${OTPROOT}/lib/erl_interface-3.6.5/include -I${YAJLROOT}/include
➥ -L${OTPROOT}/lib/erl_interface-3.6.5/lib -L${YAJLROOT}/lib
➥ ./c_src/jp_driver.c -lei_st -lyajl
```

在Mac OS X下，需要将-shared替换成-bundle -flat_namespace -undefined

suppress:

```
gcc -o ./priv/jp_driver.so
➥ -fpic -bundle -flat_namespace -undefined suppress
➥ -I${OTPROOT}/erts-5.7.5/include
➥ -I${OTPROOT}/lib/erl_interface-3.6.5/include -I${YAJLROOT}/include
➥ -L${OTPROOT}/lib/erl_interface-3.6.5/lib -L${YAJLROOT}/lib
➥ ./c_src/jp_driver.c -lei_st -lyajl
```

请拿上述命令与12.2.3节中给出的命令行做一个比较。除了源文件和目标文件（分别是c_src/jp_driver.c和priv/jp_driver.so）有所不同以外，还多出了`-fpic`和`-shared`参数，以及erl_driver.h头文件所在的包含路径${OTPROOT}/erts-5.7.5/include。.so是大部分类UNIX平台下共享库的标准扩展名，Windows下则是.dll。

链入式驱动代码的C语言部分就此迁移完毕。整个过程主要是为了修改驱动程序与Erlang VM间的数据交换手段而编写了一些包装代码。接下来，还需要对Erlang代码作出一些调整。

12.3.4 驱动的Erlang部分

为了使用链入式驱动，Erlang部分也需要作出一些修改。OTP应用的总体结构不用变，但用于管理端口的gen_server必须要改。

当前的jp_server（参见12.2.1节）调用了`open_port({spawn, Name}, Options)`，其中`Name`是某个可执行文件的路径。随后，Erlang VM将负责启动外围程序并将它的标准输入输出流与端口对接，一旦发现外围程序终止就关闭端口。很简单。

在使用链入式驱动时，启动端口的命令没有变，但C库的加载和链接却得由你来保证了。用于加载和打开端口的代码如下：

```
case erl_ddll:load(PrivDir, "jp_driver") of
    ok    -> ok;
    Other -> exit(Other)
end,
open_port({spawn, "jp_driver"}, [binary])
```

此处的要点在于共享库文件的定位和加载。和此前的外部可执行程序一样，应该将库文件安置在应用的priv目录（或其子目录）下。有了这一约定，只需调用`code:priv_dir(AppName)`便能迅速定位目标目录，参见12.2.1节。

加载共享库时调用的是`erl_ddll:load(Path, Name)`（注意有两个d），其中`Path`是库文件所在目录的路径，`Name`是*无扩展名*（通常是.so或.dll）的文件名。文件名必须与`ErlDrvEntry`结构体中定义的`driver_name`相同。如果调用返回`ok`，就表示一切正常，可以继续打开端口。对于链入式驱动来说，`exit_status`端口选项没有意义（用于处理退出状态码消息的`handle_info`回调也没必要了）。同样，`{packet,N}`选项也没用了——erl_driver API已经将数据的长度告诉你了。除了这些以外，`open_port`调用看起来与此前针对外围程序的版本并没有什么两样，只是在调用参数中指定的名称不再是文件路径，而是用于标识待加载驱动的字符串。

按12.3.3节的描述编译好C代码，就可以开始测试链入式驱动了。运行Erlang、启动json_parser

12

应用、调用`json_parser:parse_document/1`，整个过程与12.2.3节如出一辙。作为用户，除了执行速度有所提升以外，应该感觉不到任何差异。

至此，原本实现成外围程序的JSON解析器就被完全改造成链入式驱动了。通信不再需要标准I/O，而是通过C API调用直接完成数据交换。这种方式通过减少跨进程数据传递有效地提升了Erlang与外围代码间的通信效率。速度的问题终于解决了，Erlware的同仁们非常满意。

12.4　将解析器实现为 NIF

NIF在Erlang中还是个新事物，跟端口类接口有所不同。和端口驱动一样，它们采用以回调为基础的C API；但在Erlang看来它们跟普通Erlang函数没有什么区别，跟端口没有任何关系。`erl_nif` API自有一套用在Erlang和C之间传递数据结构的函数，并不使用外部项式格式。因此端口驱动代码中可重用于NIF的部分相对较少。带来的收益则是速度的提升和实现的简化。

在本节的示例中，你将用NIF接入JSON解析器库。为了把握NIF的概念，我们先来实现Erlang这边的NIF代码。

12.4.1　NIF的Erlang部分

在用NIF实现JSON解析器接口时，不再需要用gen_server打开端口——所有功能将由json_parser模块直接提供。使用之前也没必要启动应用，因此也不需要监督树。jp_app、jp_sup和jp_server这几个模块都可以省了。该应用就是一个简单的库应用，ebin/json_parser.app可以简化如下：

```
{application, json_parser,
 [{description, "JSON parser (using NIFs)"},
  {vsn, "0.1.0"},
  {modules, [json_parser]},
  {applications, [kernel, stdlib]}
 ]}.
```

所有Erlang代码都位于json_parser模块内，所需完成的功能也很简单。首先，必须实现一个用于加载共享对象的函数。如下所示：

```
init() ->
    case code:priv_dir(?APPNAME) of
        {error, _} ->
            error_logger:format("~w priv dir not found~n", [?APPNAME]),
            exit(error);
        PrivDir ->
            erlang:load_nif(filename:join([PrivDir, "jp_nifs"]), 0)
    end.
```

调用`erlang:load_nif(Path, LoadInfo)`将加载并链接Path（无文件扩展名）指定的共享库文件，然后就可以像调用普通Erlang函数一样调用NIF了。`LoadInfo`参数将被传递给`load`回调（稍后解释），可用于处理版本升级等事务。在本例中，传入0即可。

不过，如果每次调用NIF函数之前都得先调用`init()`那可就太糟糕了。是否使用NIF是函数

自身的实现细节，对用户来说应该是不可见的。好在这个问题可以用与erl_nif API同期加入Erlang的-on_load(...)模块属性来解决。以下声明表示每当Erlang VM加载该模块时都应自动调用init/0函数：

```
-on_load(init/0).
```

这样就可以保证所属模块一经加载NIF就自动可用，用户不会发现函数的真身是NIF。

最后，还得为NIF导出一个桩函数——在本例中这个函数就是parse_document/1：

```
parse_document(Data) ->
    erlang:nif_error(nif_not_loaded).
```

NIF库加载后，对应的NIF实现将会替换上述的Erlang版本。

在桩函数中调用 erlang：nif_error/1

此处调用内置函数 erlang:nif_error/1 有两个目的：首先，如果 NIF 库尚未加载就调用了 NIF 函数，它会抛出运行时错误。在这种情况下，erlang:nif_error/1 与 erlang:error/1 完全一样。其次，是为了提示 Dialyzer 等代码分析工具这段 Erlang 代码并不代表该函数的实际行为（要等到 NIF 库加载之后才能得以体现）。如果不这么做，Dialyzer 会认为这个函数总是抛出异常从而就该函数发出警告。基于上述原因，实现 NIF 桩函数时应该调用 erlang:nif_error/1。

准备就绪后，剩下的工作就是构建由json_parser:init()加载的共享库priv/jp_nifs.so了。

12.4.2 NIF的C代码部分

请将12.3节中实现的c_src/jp_driver.c复制一份，更名为c_src/jp_nifs.c。虽有不少雷同之处，但还是有不少地方要改。

首先，文件开头处包含的erl_driver.h和ei.h应换成erl_nif.h：

```
#include <erl_nif.h>
```

drv_start、drv_stop、drv_output、make_error函数和jp_driver_entry、drv_data_t结构体，还有DRIVER_INIT宏，统统删掉。

1. ErlNifEnv环境

在链入式驱动中，实例专有信息必须保持在自定义的结构体中。而使用NIF时，Erlang VM会向你的C函数传入一个被视作句柄的ErlNifEnv对象指针，后续用到的大部分erl_nif API函数都会用上这个句柄。（有关erl_nif API的详情请参见Erlang/OTP文档中的ERTS参考手册。）为了将该句柄传入YAJL解析器回调函数，就必须把它存入state_t结构体。出于某种原因（后续说明），状态结构体中还得保存一个标志变量，用于指示YAJL解析器解析的上一个元素是否为某键/值对中的键。因此，state_t结构体的定义中得加上这么两行：

```
ErlNifEnv *env;
int key;
```

12

另外，由于不再需要调用ei库来构建项式，请删除下面这行

```
ei_x_buff x;
```

详情见下。

2. 内存管理

在链入式驱动中，必须用erl_driver库提供的driver_alloc等函数来分配内存。在NIF中，则必须换用erl_nif库提供的enif_alloc等函数。这些函数的第一个参数都是ErlNifEnv，因此必须将它设置为内存分配函数的上下文。请在函数parse_json中调用yajl_alloc之前的位置加上以下代码：

```
alloc_funcs.ctx = st->env;
```

（alloc_funcs结构体中的这个字段原本是NULL。）

> **注意**　直到Erlang/OTP R14为止NIF API才最终定稿。本书讲解的是R13版本。二者之间的主要区别在于最新版本中enif_alloc等函数不再以ErlNifEnv指针作为第一个参数。使用R14时，应删除该参数。

接下来，分别用以下调用替换YAJL内存分配包装函数中的driver_alloc、driver_realloc和driver_free：

```
enif_alloc((ErlNifEnv *)ctx, sz)
enif_realloc((ErlNifEnv *)ctx, ptr, sz)
enif_free((ErlNifEnv *)ctx, ptr)
```

YAJL解析器回调函数中的内存分配逻辑也要作出相应修改，不过这几个函数后续几乎要完全重写，这些小的修改此处就不再赘述了。

3. 跟踪容器内容

用erl_nif库替换ei库并不简单，没有了ei中的ei_x_buff，就无法增量构建用于表示数组和映射的Erlang项式了。用erl_nif创建列表和元组时，必须事先将项式的大小告知相关函数；代码清单12-7中那种事后补填元素数目的手法在此失效。不过办法还是有的：可以先将项式存入C数组，然后用erl_nif中的enif_make_tuple_from_array和enif_make_list_from_array函数一次性将C数组转换成所需的元组或列表。这样一来，只需要在JSON容器的解析过程中做好相关数组的维护工作即可。为此，container_t结构体中还需要再加两个字段，分别是arraysz和array：

```
typedef struct container_t {
  int count;     /* number of elements */
  int arraysz;  /* size of elements array */
  ERL_NIF_TERM *array;  /* elements array */
  struct container_t *next;
} container_t;
```

另外，index字段已经没用了，可以删除。

4. NIF实现函数

现在我们来看用于实现NIF的C函数。由于对应的Erlang函数名为parse_document/1（位于json_parser模块内），因此我们将该C函数取名为parse_document_1。该函数将取代先前版本中的process_data函数。相关代码如代码清单12-11所示，其中也包括了用于将NIF接入Erlang VM的声明。

代码清单12-11　c_src/jp_nifs.c：NIF实现函数

```
static ERL_NIF_TERM parse_document_1(ErlNifEnv *env, int argc,
                                     const ERL_NIF_TERM argv[])
{
  state_t st;                                        ❶ 在解析状态中保存NIF环境
  st.env = env;                                    <-
  st.key = 0;
  ERL_NIF_TERM term;                               ❷ 设置虚拟的顶层容器
  container_t c = { 0, 1, &term, NULL };
  st.c = &c;

  if (argc != 1 || !enif_is_binary(env, argv[0]))
    return enif_make_badarg(env);

  ErlNifBinary bin;                                  ❸ 获取数据的地址和长度
  if (!enif_inspect_binary(env, argv[0], &bin))    <-
    return enif_make_badarg(env);
  const char *err;                                   运行解析器 ❹
  if ((err = parse_json(&st, bin.data, bin.size)) != NULL) {   <-
    return enif_make_tuple2(env, enif_make_atom(env, "error"),
                            enif_make_string(env, err, ERL_NIF_LATIN1));
  }
  return enif_make_tuple2(env, enif_make_atom(env, "ok"), term);
}
static ErlNifFunc json_parser_NIFs[] = {            <-
  {"parse_document", 1, &parse_document_1}           ❺ 罗列所有NIF
};

ERL_NIF_INIT(json_parser, json_parser_NIFs, NULL, NULL, NULL, NULL);
```

所有NIF实现函数的签名都是一样的：接受 3 个参数，并返回一个ERL_NIF_TERM对象（在erl_nif API中定义）。第一个参数env就是前面提到过的ErlNifEnv指针。将之存入state_t结构体以便后续访问❶。第二个参数argc是Erlang调用NIF时传入的参数数目。（这样便可以用单个C函数实现多个Erlang NIF。）最后，数组argv中的元素就是传入的各个参数（数目与argc中的一致）。

此前，解析结果是用ei库的ei_x_buff来构建的，NIF则不同。NIF函数的返回值类型是用于表示Erlang返回数据的ERL_NIF_TERM。为了保存JSON解析回调返回的解析结果项式，需要事先设置一个虚拟的顶层container_t结构体❷。该结构体中仅有一个项式，由变量term保存，相当于一个仅含单个元素的C数组；最后，一切正常的话，将term中的值装入二元组{ok, ...}返回。

一般来说，在执行具体功能之前，应该先对NIF函数参数的数量和数据类型进行检查。本例中（唯一的）参数应该是个二进制串；如果不是，就应该返回一个专门用于触发badarg异常的

ERL_NIF_TERM值来通报运行时错误。

在开始解析之前，必须先从二进制串中找出实际的JSON数据。调用enif_inspect_binary并传入一个ErlNifBinary结构体，便可取出待解析的JSON数据❸。有了这个ErlNifBinary结构体，就可以像之前一样调用parse_json函数了❹。如果解析结果是个错误字符串，则向Erlang返回{error, String}元组。

5. 注册NIF

处理完上述事务，还得准备一个ErlNifFunc数组，这样Erlang VM才能获知库中实现的NIF。数组中应逐一填入各NIF的Erlang函数名和元数以及对应的C实现函数❺。此外，还必须调用ERL_NIF_INIT宏将该数组连同这批NIF函数所属模块的模块名一同告知Erlang VM。（请注意，模块名json_parser不带引号。）

ERL_NIF_INIT的后4个参数是指向NIF生存期管理函数的函数指针，可以按需选用。（本例中全部置为NULL即可。）按照在ERL_NIF_INIT宏中的出现次序，这4个函数分别是load、reload、upgrade和unload。其中load在NIF被载入系统时调用，unload在NIF即将从系统中卸载时调用；reload函数在NIF重新载入时调用，upgrade函数则在对NIF函数执行运行时代码升级时调用。在本例中，我们暂时用不上这些函数。

6. 重写YAJL解析器回调

将JSON解析器改造成NIF的最后一步就是重写YAJL的解析器回调函数。此前我们采用ei库将解析结果编码成Erlang数据，现在则必须换用工作方式迥异的erl_nif API。因此大部分回调代码都没法重用，必须重写。所幸，这些改动多半都还算简单，差别较大的主要是handle_map_key、handle_start和handle_end函数。

在跟踪容器中的元素时，策略也得改改：计数就不必了，但需要把每个元素都存入自行分配的临时数组。count_element工具函数将被替换成更为复杂的add_element函数，该函数以解析器状态和待添加的项式为输入参数，如代码清单12-12所示。

代码清单12-12　c_src/jp_nifs.c：add_element工具函数

```
static void add_element(state_t *st, ERL_NIF_TERM t)
{
  container_t *c = st->c;
  if (c != NULL) {
    if (c->count >= c->arraysz) {
      c->arraysz *= 2;
      c->array = enif_realloc(st->env, c->array, c->arraysz);
    }
    if (st->key) {
      c->array[c->count-1] =
          enif_make_tuple2(st->env, c->array[c->count-1], t);
      st->key = 0;
    } else {
      c->array[c->count] = t;
      ++(c->count);
    }
  }
}
```

❶ 调整数组大小
❷ 处理完整的键/值对
❸ 处理其他元素

起初，每个容器中的元素数都是零，并且附带一个较小的预分配的数组用于存储加入容器的元素。当数组不足以容纳新加入的元素时，对数组进行扩容❶。接下来分两种情况，如果新加入的元素是键/值对中的值（可通过检查`st->key`字段得知），则上一个加入数组的元素是对应的键，这时应该将键从数组中取出，然后键和值重组为一个二元组，再放回数组。这种情况下元素计数不应再次递增，否则就重复了，不过必须重置key标志位❷。`st->key`标记为零的情况就简单多了：只需将项式插入数组当前位置并递增元素计数即可❸。

有了这一机制，用`erl_nif` API重写YAJL回调就简单多了，如代码清单12-13所示。

代码清单12-13 c_src/jp_nifs.c：新版的简单YAJL回调

```
static int handle_null(void *ctx)
{
  state_t *st = (state_t *)ctx;
  add_element(st, enif_make_atom(st->env, "undefined"));
  return 1;
}

static int handle_boolean(void *ctx, int boolVal)
{
  state_t *st = (state_t *)ctx;
  if (boolVal) {
    add_element(st, enif_make_atom(st->env, "true"));
  } else {
    add_element(st, enif_make_atom(st->env, "false"));
  }
  return 1;
}

static int handle_integer(void *ctx, long integerVal)
{
  state_t *st = (state_t *)ctx;
  add_element(st, enif_make_long(st->env, integerVal));
  return 1;
}

static int handle_double(void *ctx, double doubleVal)
{
  state_t *st = (state_t *)ctx;
  add_element(st, enif_make_double(st->env, doubleVal));
  return 1;
}

static int handle_string(void *ctx, const unsigned char *stringVal,
                         unsigned int stringLen)
{
  state_t *st = (state_t *)ctx;
  ErlNifBinary bin;
  enif_alloc_binary(st->env, stringLen, &bin);
  strncpy((char *)bin.data, (char *)stringVal, stringLen);
  add_element(st, enif_make_binary(st->env, &bin));
  return 1;
}
```

```
static int handle_map_key(void *ctx, const unsigned char *stringVal,
                          unsigned int stringLen)
{
  state_t *st = (state_t *)ctx;
  ErlNifBinary bin;
  enif_alloc_binary(st->env, stringLen, &bin);
  strncpy((char *)bin.data, (char *)stringVal, stringLen);
  add_element(st, enif_make_binary(st->env, &bin));
  st->key = 1;
  return 1;
}
```

❶ 注明插入了一个键

仔细观察就会发现，这些函数都遵循相同的模式：先用enif_等函数创建Erlang项式再调用add_element将之插入当前容器。你应该还记得，在parse_document_1函数（代码清单12-11）中曾经创建了一个虚拟的顶层容器，因此不用担心没有容器保存这些元素。比较值得注意的是handle_map_key函数，它首先将键当作普通字符串插入，然后再做好标记，注明刚刚插入的是键/值对中的键，这样下一个元素才能得到正确的处理❶。

最后，代码清单12-14展示了新版本的JSON容器回调，这些回调分别负责处理JSON数组和映射的头尾。出人意料的是，作为本章的最后一段代码，这几个函数极其简单明了。

代码清单12-14　c_src/jp_nifs.c：新的YAJL容器回调

```
static int handle_start(void *ctx, int array)
{
  state_t *st = (state_t *)ctx;
  container_t *c = enif_alloc(st->env, sizeof(container_t));
  c->next = st->c;
  st->c = c;
  c->count = 0;
  c->arraysz = 32;  /* initial term buffer size */
  c->array = enif_alloc(st->env, c->arraysz);
  return 1;
}

static int handle_start_map(void *ctx)
{
  return handle_start(ctx, 0);
}

static int handle_start_array(void *ctx)
{
  return handle_start(ctx, 1);
}

static int handle_end(void *ctx, int array)
{
  state_t *st = (state_t *)ctx;
  container_t *c = st->c;
  st->c = c->next;
  if (array) {
```

❶ 分配容器结构体、将之链入解析器状态并完成初始化

❷ 从解析器状态中摘除容器

```
    add_element(st, enif_make_tuple_from_array(st->env, c->array,
                                               c->count));
  } else {
    add_element(st, enif_make_list_from_array(st->env, c->array,
                                              c->count));
  }
  enif_free(st->env, c);
  return 1;
}
static int handle_end_map(void *ctx)
{
  return handle_end(ctx, 0);
}
static int handle_end_array(void *ctx)
{
  return handle_end(ctx, 1);
}
```

❸ 创建并添加Erlang列表/元组

解析 JSON数组或映射的开头时，handle_start会（调用enif_alloc）分配一个新的容器结构体，将之链入解析器状态，并完成初始化。它还会预先分配一个相对较小的数组，用于存储容器中的元素，必要时add_element还会调整数组的大小❶。

调用handle_end时，JSON数组或映射中的元素已经全部装入容器结构体中的C数组。接下来，将该容器从解析器状态中摘除，这样后续新的元组或列表才能进入外层容器❷。然后，调用enif_make_tuple_from_array或enif_make_list_from_array将C数组转换成Erlang项式❸，再将之放入新的当前容器。至此，先前使用的容器结构体就可以释放了。

12.4.3 编译与运行代码

请用以下命令行调用gcc编译本节中的C代码：

```
$ gcc -o ./priv/jp_nifs.so -fpic -shared -I${OTPROOT}/erts-5.7.5/include
➥ -I${YAJLROOT}/include -L${YAJLROOT}/lib ./c_src/jp_nifs.c -lyajl
```

和jp_driver一样，Mac OS X下的命令行参数略有不同：

```
gcc -o ./priv/jp_nifs.so -fpic -bundle -flat_namespace -undefined suppress
➥ -I${OTPROOT}/erts-5.7.5/include
➥ -I${YAJLROOT}/include -L${YAJLROOT}/lib ./c_src/jp_nifs.c -lyajl
```

和端口驱动代码的处理方法类似，NIF也要编译成共享库。与12.3.3节中的编译命令相比，除了源文件和目标文件不同以外，另一个差异就是这里省去了erl_interface库的头文件路径和库文件路径。待priv/jp_nifs.so就绪后，12.4.1节中的代码应该就可以工作了。正如我们在本节中曾经说明过的那样，现在用于对接JSON库的代码全都包含在json_parser模块内了，模块加载时会自动设置并加载NIF库。使用NIF函数前无须启动任何应用，如下所示：

```
$ erl -pa ../json_parser/ebin
...
1> Doc = <<"[null, true, {\"int\": 42, \"float\": 3.14}]">>.
...
```

```
2> json_parser:parse_document(Doc).
{ok,{undefined,true,[{<<"int">>,42},{<<"float">>,3.14}]}}
3>
```

至此，本章就要告一段落了。我们介绍了 3 种在Erlang中集成外围代码的方法，并讨论它们各自的优缺点。就NIF而言，一大缺点就在于并发性不足：NIF调用会阻塞发起调用的调度器，直至NIF返回为止。如果你用的是典型的每个CPU一个调度器的Erlang系统，那就意味着在NIF调用结束前对应的CPU无法运行其他Erlang进程。如果只是直接调用外围代码写成的库而且能够快速返回，用NIF就再适合不过了。至于JSON解析器是不是适合采用NIF，这就要看应用场景了：在解析大型文档时，系统的响应度会有所降低，但吞吐量应该会很好。

12.5　小结

我们在本章中讨论了 3 种Erlang与外围代码通信的基本方法。从安全性出发，通过端口接入外围程序是最为稳妥的方案，而且无须开发额外的C代码就可以直接调用大部分UNIX程序，只是在部分情况下性能方面可能会拖后腿。如果确实有必要追求速度，那么可以选用链入式驱动或NIF，至于到底该用哪个，这取决于更深层次的需求——比如C代码中是否会出现异步IO操作等。

不过并不是所有人都会C，近年来Java的应用也颇为广泛。要是能够绕开C，直接用Java与Erlang通信，那该多好。在下一章中，我们将对另一种外围代码通信方式进行探讨，它就是Java节点。

第13章

用Jinterface实现Erlang和Java间的通信

13

本章概要

- 用Jinterface与Java程序通信
- 桥接Erlang与HBase数据库之间的桥梁
- 将HBase集成至Simple Cache应用中

在上一章中，我们探讨了如何利用Erlang端口来集成外围代码。这种方法既实用又通用，但却未必是最为便利的手段。本章我们将另辟蹊径，将外围代码伪装成一个Erlang节点，直接采用Erlang分布式协议（参见第8章）与Erlang通信。好在Erlang语言的开发者们已经做了大量铺垫，为C和Java节点的开发提供了良好的库支持，使得这部分工作得以简化。

经过本书第2部分的打磨，我们开发的缓存应用已经可以适用于企业应用环境了，越来越多的个人和组织开始将它应用到自己的项目之中。其中有这么一个小组，他们的数据不仅需要在Mnesia的内存表中保存，而且还要持久化保存。按照他们的需求，凡是插入缓存的数据基本上都得永久保存。磁盘型Mnesia表固然是一个选择，但在潜在的巨大数据量面前，这个方案似乎并不恰当。为了解决这个问题，我们决定利用缓存外部的HBase集群来存储缓存数据。

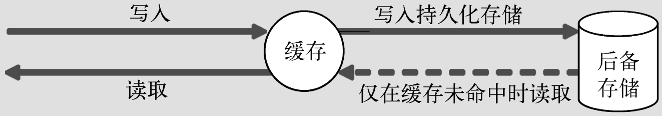
本章我们先大致讲解一下HBase的工作方式，然后学习Jinterface的工作原理并实现一个基本的Java节点示例。在本章的其余内容中，我们将以这些经验为基础将Java接入Erlang，进而实现基于HBase表的Erlang项式存取，最后再将这些成果集成进Simple Cache应用。

HBase

HBase是出自Hadoop项目的一个数据库（http://hadoop.apache.org/hbase/）。它的设计源自Google的Bigtable数据库，为大数据集提供了快捷可靠的存储支持。HBase十分健壮，存储模型和分布式模型也很清晰。集成了它，用户存多少数据都没问题。

缓存与HBase系统的交互过程大致如下：当缓存收到查询请求时，首先检查Mnesia表中是否存在待查数据。如果存在，就直接返回数据；否则便尝试从HBase中读取数据、插入Mnesia，再返回。当收到写请求时，缓存会将数据同时写入HBase和Mnesia。这一版的缓存应用不仅

为用户提供了一套高效的前端内存缓存，还提供了一套可靠的后备存储系统。二者之间的关系如下图所示。



缓存位于持久化存储的前端。处理读操作时尽可能只访问缓存，处理写操作时则同时将数据写入后备存储系统

集成 HBase 的手段有很多，方案之一便是通过 HTTP 访问 HBase 的 REST 式 API。不过，在此我们打算利用 Jinterface 库直接访问 HBase 的 Java API。Jinterface 是一个 Java 库，它能够将 Java 应用伪装成 Erlang 集群中的节点。

本书关注的是 Erlang 和 OTP，而非 Java 或 HBase。因此，我们的讨论将集中于如何将 HBase 节点用作缓存的存储后端，而不会深入讨论 HBase 或 Java 语言本身。

13.1 利用 Jinterface 在 Erlang 中集成 Java

在深入探讨HBase及其集成方案之前，我们先来看一看Jinterface。Jinterface是一个Java库，旨在让Java也能访问Erlang的分布式层。它并没有照搬Java的传统思路，而是尽可能原汁原味地保留了Erlang的模型。对我们这些从Erlang转入Java的人来说，这可是个利好消息：从节点到信箱再到元组和原子等各种更细粒度的对象，Erlang中的每种结构几乎都有一个Java类与之对应。我们将从中挑选一些最为关键的类进行说明，看看它们到底是如何映射到Erlang中去的。

13.1.1 `OtpNode`类

Erlang的分布式模型以相互连接的节点为基础。在Jinterface库中，节点由`OtpNode`类表示。节点对象可以与其他节点（不一定是真正的Erlang节点）连接和交互。跟普通Erlang节点一样，用Jinterface实现的节点也有节点名，并且也支持cookie认证机制（参见8.2.4节）。创建`OtpNode`对象即可启动Java节点：

```
OtpNode node = new OtpNode("myJavaNode");
```

就这么简单。如果字符串包含字符`@`，那便是完整节点名，譬如`"myNode@frodo.erlware.org"`。否则，Jinterface会在名称字符串之后追加字符`@`和本地主机名，从而构成短节点名，比如`myNode@frodo`。（我们曾经提过，Erlang集群中相互连接的各个节点必须用同一种命名方式，要么用短节点名，要么用长节点名，这两种方式分别对应于启动命令中的`-sname`和`-name`命令行参数。）编译代码的方法留待13.1.4节介绍。

如果需要设置用于认证外来连接的cookie（参见8.2.4节），创建节点时还得再加一个参数：

```
OtpNode node = new OtpNode("myJavaNode", "secretcookie");
```

`OtpNode`类隐藏了节点的底层通信和连接管理等技术细节，有了它，在Erlang集群中集成Java

代码可就简单多了。不过，要想让节点发挥作用，还得给它配备一个信箱。

13.1.2 `OtpMbox`类

有了信箱，`OtpNode`节点才能与集群中的其他节点交互。它们的作用类似于Erlang进程的信箱，但却不属于任何进程。仅从通信角度考虑，在Jinterface的模型中信箱标识符的作用与进程标识符类似——即用作消息传递的目标地址。Jinterface并不负责线程管理，而是全权委托给了开发者，此外还允许直接访问信箱，因此线程之间也可以采用消息传递进行通信。

信箱对象由`OtpMbox`对象创建。如果需要，可以给信箱命名。具名信箱会在本地节点（即Java节点）上注册，类似于Erlang节点上的注册进程。具名信箱可通过名称接收消息，这与经注册的Erlang进程一样；匿名信箱则只能通过pid或信箱对象的引用才能访问。具名信箱的创建方法如下所示：

```
OtpMbox named_mbox = node.createMbox("myNamedMbox");
```

创建匿名信箱则更为简单：

```
OtpMbox anon_mbox = node.createMbox();
```

有了信箱，便可以收发消息了。这里我们将用上`OtpMbox`类提供的两个基础API：`send`和`receive`。这两个API还有多个变体，详情请参阅Javadoc文档（参见www.erlang.org/doc/apps/ jinterface/java/）。

俗话说入乡随俗，在Java和Erlang间传递数据当然还得用Erlang的格式。好在Jinterface已经为我们做好了万全的准备。

13.1.3 Erlang数据结构的Java映射

节点相互收发消息时，消息中的数据必须用Jinterface提供的类型映射类来表示。表13-1中列出的就是各种Erlang数据类型在Java中的映射类。其中`OtpErlangObject`是所有Erlang类型映射类的父类。

表13-1 Jinterface中用于表示Erlang数据的Java类

Erlang类型	Java类
原子	`OtpErlangAtom`、`OtpErlangBoolean`
二进制串、位串	`OtpErlangBinary`、`OtpErlangBitstr`
Fun函数	`OtpErlangFun`、`OtpErlangExternalFun`
浮点数	`OtpErlangDouble`、`OtpErlangFloat`
整数	`OtpErlangInt`、`OtpErlangLong`、`OtpErlangShort`、`OtpErlangChar`、`OtpErlangByte`、`OtpErlangUShort`、`OtpErlangUInt`
列表	`OtpErlangList`、`OtpErlangString`
pid	`OtpErlangPid`
端口	`OtpErlangPort`
引用	`OtpErlangRef`
元组	`OtpErlangTuple`
项式	`OtpErlangObject`

为了说明这些类的用法，下面我们来看几个示例。首先以下列Erlang项式为例：

```
{some_atom, "Some string", 22}
```

以下代码负责将项式映射到Java，再用匿名信箱将之发送至指定的具名信箱。唉，就这么点儿逻辑，用Java来写还真是麻烦：

```
OtpErlangAtom   anAtom   = new OtpErlangAtom("some_atom");
OtpErlangString aString = new OtpErlangString("Some string");
OtpErlangInt    anInt    = new OtpErlangInt(22);

OtpErlangTuple aTuple =
  new OtpErlangTuple(new OtpErlangObject[]{anAtom, aString, anInt});

anon_mbox.send("myNamedMbox", aTuple);
```

可以看出，Erlang数据类型与Java类之间的映射颇为直截了当。需要特别注意的是，在构建元组、列表等复合容器对象时应循序渐进，必须从容器内的单个对象入手逐步构建。

现在，再反过来看看如何将Erlang数据转换成普通的Java对象。假设我们的具名信箱收到了一条消息（简单起见，此处略去了消息结构分析，假定你预先知晓消息结构），消息解析过程如下：

```
OtpErlangObject msg = named_mbox.receive();

OtpErlangTuple t = (OtpErlangTuple) msg;

String theAtom   = ((OtpErlangAtom)   t.elementAt(0)).atomValue();
String theString = ((OtpErlangString) t.elementAt(1)).stringValue();
int    theInt    = ((OtpErlangInt)    t.elementAt(2)).intValue();
```

如果信箱中存在未读消息，`receive()`方法将返回信箱中的第一条消息，否则该方法将持续阻塞直至有新消息抵达为止。该方法的调用结果是一个`OtpErlangObject`对象，可进一步转换为更加特化的类型。只要知道收到的是什么类型的数据，就不难用Jinterface将之转换成原生Java数据。（究竟该转换成什么类型，取决于具体的数据处理方式，这就要看你了。）

本章的目标是用Jinterface在Simple Cache和HBase之间建立一个接口层，如图13-1所示。为此，必须先搞清楚Erlang和Java间的通信方式。后续两节将用一个完整示例分别展示从Java到Erlang以及从Erlang到Java的通信过程。

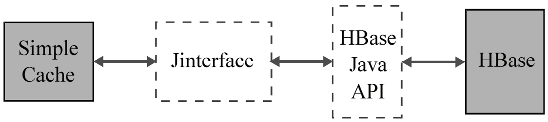


图13-1　本章的终极目标：让Simple Cache通过Jinterface和HBase Java API访问HBase表

13.1.4　示例：Java中的消息处理

以下示例代码的功能如图13-2所示：首先接收一个内含姓名字符串和发送方pid的二元组，再向发送方回复一个结构类似的二元组，其中包含自身信箱的pid和一句问候语。

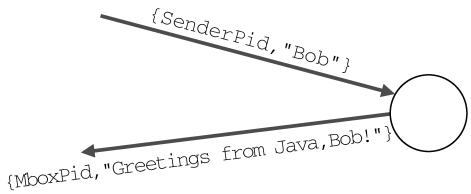


图13-2 Jinterface示例的通信模式。收到内含名称字符串的消息后，Java代码将发回一句问候

请新建源文件JInterfaceExample.java并录入示例代码。我们先来看下该如何编译，然后再着手分析代码。

1. 编译Java程序

Java程序的编译过程就是用javac将.java文件转换为.class文件的过程。类似于erl的-pa参数，java和javac采用-cp（class path）参数来指定代码搜索路径。在Java中，无论是编译程序还是运行.class文件，都必须给出.class文件的搜索路径。（相较之下，编译Erlang代码时之所以要指定代码搜索路径只是为了辅助编译器检查行为模式声明，并不是必需的。）

javac需要知道到哪儿才能找到随Erlang/OTP一并安装到本地的Jinterface库。该路径一般指向/usr/local/lib/erlang/lib/jinterface-1.5.1/priv/OtpErlang.jar等位置。在继续学习之前请先在自己的电脑上找到对应的OtpErlang.jar文件。

找到对应的路径之后，请用以下参数调用javac：

```
$ javac -cp /path/to/OtpErlang.jar JInterfaceExample.java
```

一切正常的话，当前目录下应该会多出一个名为JInterfaceExample.class的文件。如果命令执行失败，报错声称javac not found，那么大概是你的机器上没装Java Development Kit（JDK）的缘故。请从http://java.sun.com/下载并安装JDK，然后确保javac位于系统搜索目录下，并将JAVA_HOME环境变量设置为Java安装目录的路径。

2. Java示例代码

OTP Jinterface库的代码位于com.ericsson.otp.erlang包内。使用它之前，必须先导入这个包：

```
import com.ericsson.otp.erlang.*;
```

接下来，定义JInterfaceExample类：

```
public class JInterfaceExample {
  // all the rest of the code goes here
}
```

其余代码都位于{和}之间。首先是作为整个程序入口点的main()方法，它负责创建Jinterface-Example类的实例并用命令行参数调用该实例的process()方法：

```
public static void main(String[] args) throws Exception {
  if (args.length != 3) {
    System.out.println("wrong number of arguments");
    System.out.println("expected: nodeName mailboxName cookie");
    return;
```

13

```
    }
    JInterfaceExample ex = new JInterfaceExample(args[0],args[1],args[2]);
    ex.process();
  }
```

然后，添加两个成员变量，分别用于表示节点和信箱：

```
private OtpNode node;
private OtpMbox mbox;
```

节点的初始化由`JInterfaceExample`的构造函数负责，代码如下：

```
public JInterfaceExample(String nodeName, String mboxName, String cookie)
throws Exception {
  super();
  node = new OtpNode(nodeName, cookie);
  mbox = node.createMbox(mboxName);
}
```

首先用指定的节点名和安全cookie创建节点。然后利用节点对象创建具名信箱。最后，实际工作交由`process()`方法完成，它负责处理消息收发，如代码清单13-1所示。

代码清单13-1　Jinterface消息处理示例

```
private void process() {
  while (true) {
    try {
      OtpErlangObject msg = mbox.receive();
      OtpErlangTuple t = (OtpErlangTuple) msg;                        分解元组 ❶
      OtpErlangPid from = (OtpErlangPid) t.elementAt(0);
      String name = ((OtpErlangString) t.elementAt(1)).stringValue();
      String greeting = "Greetings from Java, " + name + "!";
      OtpErlangString replystr = new OtpErlangString(greeting);
      OtpErlangTuple outMsg =
        new OtpErlangTuple(new OtpErlangObject[]{mbox.self(),
                                                  replystr});
      mbox.send(from, outMsg);
    } catch (Exception e) {                                           创建应答元组 ❷
        System.out.println("caught error: " + e);
    }
  }
}
```

该方法的主体是一个`while (true) {...}`无穷循环，消息处理便在该循环内完成。这里的错误处理逻辑非常简单，期间出现任何错误，只会打印错误信息，然后便继续执行。收到的消息应该是个二元组，其中包含发送方的pid和一个姓名字符串。接下来的代码负责分解该元组❶并构造出应答字符串，然后将之与信箱pid一起放入应答元组❷。这些操作在上一节中我们都介绍过。

成功编译JInterfaceExample.java后，下面我们就来运行这个示例。

13.1.5　在Erlang中与Java节点通信

如8.2.3节所述，Erlang节点需要依靠EMPD守护进程来定位其他节点。基于Jinterface库的节点也不例外，但Jinterface自身无法启动EPMD。不过Erlang节点启动时会对本地的EPMD进程进行

检查。如果EPMD进程不存在，Erlang节点会自动启动EPMD。这样一来问题就简单了：启动Jinterface代码之前先启动一个Erlang节点就万事大吉了。（启动之后这个Erlang节点就没用了，即便该节点退出，EPMD也会继续运行下去，直至进程终止或系统重启。）

首先以`secret`为cookie，用`-sname`方式启动一个Erlang节点：

```
$ erl -sname erlangNode -setcookie secret

Eshell V5.7.4  (abort with ^G)
(erlangNode@frodo)1>
```

接下来，在新的终端窗口中启动Java节点。搞定了这一环，在Simple Cache应用中集成HBase Java API的大业便迈出了坚实的第一步。

启动Java节点的命令行如下（同处一行），请依据实际情况酌情修改.jar文件的路径：

```
java -cp .:/path/to/OtpErlang.jar JInterfaceExample javaNode
➥theMailbox secret
```

和`javac`一样，此处也需要用`-cp`参数指定.class文件的搜索路径。（请注意，当前目录“.”也要囊括在内，这样java才能找到JInterfaceExample.class。）接着，`java`会调用包含`main()`方法的类，此处即`JInterfaceExample`。其余参数将传给`main()`，它们分别是Java节点的节点名、信箱名和cookie。

Java程序启动后，两个节点便会相互搜寻并直接通过Erlang分布式协议展开通信，如图13-3所示。

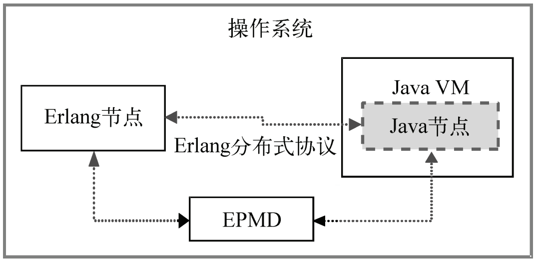


图13-3　运行在同一台机器上的Java节点和Erlang节点，二者通过同一个EPMD进程搜寻到对方，并通过Erlang分布式协议展开通信

现在返回上一个终端窗口中的Erlang节点操练一番：

```
(erlangNode@frodo)1> net_adm:ping(javaNode@frodo).
pong
(erlangNode@frodo)2> {theMailbox, javaNode@frodo} ! {self(), "Eric"}.
{<0.39.0>,"Eric"}
(erlangNode@frodo)3> receive {Mbox, Msg} -> Msg end.
"Greetings from Java, Eric!"
(erlangNode@frodo)4> Mbox.
<5569.1.0>
(erlangNode@frodo)5> Mbox ! {self(), "Martin"}.
{<0.39.0>,"Martin"}
(erlangNode@frodo)6> receive Tuple -> Tuple end.
{<5569.1.0>,"Greetings from Java, Martin!"}
```

在Erlang节点看来，Java节点和其他Erlang节点一般无二，基于普通消息传递的通信十分顺畅。请注意，在不清楚信箱的唯一标识符（参见8.2.5节）的情况下，应采用`{Name, Node} ! Message`格式的命令向信箱发送消息。等收到Java代码发回的应答后，就可直接采用应答中的标识符来发送后续的消息了。

现在，你已经掌握了让Erlang与Java节点通信的方法，可以开始开发Simple Cache与HBase之间的接口了。我们先来安装HBase，并将它配置成缓存的后备存储。

13.2　安装和配置 HBase

正式开始接口开发之前，不妨先配置好HBase，以方便后续的测试。这一节将简单介绍HBase的安装和配置。我们的主要任务是配置用于存储缓存数据的表，但首要任务是安装。

13.2.1　下载和安装

首先从http://hadoop.apache.org/hbase/下载HBase的tarball文件（参见10.3.1节）并解压到合适的位置。其次，为了编译本章的代码，还需要从http://hadoop.apache.org/common/下载Hadoop Common。编译代码时，`javac`的class path中必须同时包含这两个包中的hbase-<version>.jar和hadoop-<version>-core.jar。

HBase有个不太寻常的要求：系统中必须装有SSH服务器（sshd）。不少Linux桌面系统在默认情况下并不带sshd，必须手动安装（最好借助系统自带的标准软件包管理器来安装）。在Windows系统下，可以用Windows版的OpenSSH；倘若用Cygwin的话，不妨配置一下sshd服务。

下载HBase的tarball文件后，使用以下命令解压：

```
$ tar -xzf hbase-0.20.3.tar.gz
```

解压后，便可就地启动HBase了。进入解压HBase的目录，并执行启动脚本：

```
$ cd hbase-0.20.3
$ ./bin/start-hbase.sh
```

这时，HBase会通过SSH连接本地系统以收集所需的信息。期间会多次向你询问密码。

注意　如果HBase在启动时报出Java could not be found错误，请打开conf/hbase-env.sh文件，加上一行`export JAVA_HOME`以指明JDK的安装路径。

启动脚本执行完毕后，就可以开始进一步的配置了。

13.2.2　配置HBase

就当前的需求而言，配置工作比较简单。只需启动HBase的shell，创建一张用于存储缓存数据的表就可以了。请用以下命令启动shell（请注意`hbase`和`shell`之间的空格）：

```
$ ./bin/hbase shell
```

稍等几秒钟，紧跟在HBase当前版本相关信息之后的便是HBase的shell提示符。既然是用于存储缓存数据，那么表中保存的就应该是从唯一标识符到二进制数据块的映射表。建立这样一张表的HBase命令如下：

```
HBase (main):001:0> create 'cache', {NAME => 'value'}
0 row(s) in 2.3180 seconds
HBase (main):002:0> exit
```

这样便建好了一张名为cache的表（这是一张映射表，详情请参见HBase文档。），表中仅含一个字段，名为`value`。和大多数关系型数据库不同，在HBase中一切都是二进制数据，因此无须指定字段类型。

还挺简单的，是吧？HBase配置完毕，现在正式开始开发接口，为了能在Simple Cache中访问HBase而奋斗吧！

13.3　为 Simple Cache 和 HBase 牵线搭桥

在本章中，我们将搭建一套用于桥接Erlang与HBase的接口，这套接口是按缓存应用的后端存储需求量身定制的——我们并不打算写一套通用的HBase绑定。明确定位之后，工作量也随之骤减。东西虽小，这一方案的结构还是十分清晰的；为了让任务更富有挑战性，我们还用上了线程池，以便能够以异步方式处理请求。

这套接口分为 4 个主要部件，如图13-4所示。

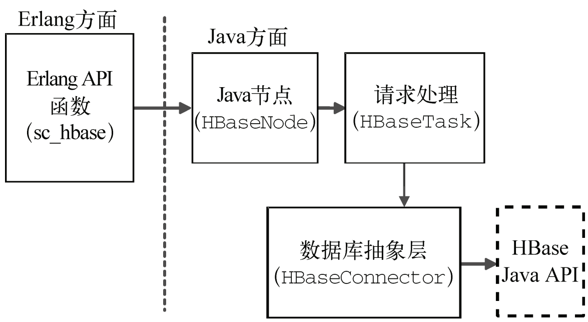


图13-4　Erlang-HBase桥接接口由1个Erlang模块和3个Java类组成，界限清楚、职责明晰

- ❑ Erlang代码——即sc_hbase模块，内含`put`、`get`、`delete`等API。和其他gen_server实现模块提供的API一样，这些函数负责封装简单消息协议。
- ❑ Java类`HBaseConnector`——通过直接调用HBase Java API来实现上述操作，隐藏相关的实现细节。
- ❑ 主Java类`HBaseNode`——用于表示Java节点。该类与13.1.4节的示例代码类似，负责处理和派发来自Erlang的请求。

13

- Java类HBaseTask——负责在各自的线程中处理请求。通过调用HBaseConnector执行相关功能后，将应答回传给Erlang客户端。

本节剩余内容会依次介绍这4个部件的实现。其中最简单的当属Erlang模块sc_hbase，不妨先拿它开刀。该模块定义了Erlang与Java间的通信协议。

13.3.1　Erlang方面：sc_hbase.erl

本章所欲达成的目标我们已在图13-1中展示过了。Simple Cache中的插入、查找、删除等主要数据库操作，都必须反映到HBase上。

按照这个方向，我们先在simple_cache应用中新建一个模块sc_hbase用作Erlang HBase API，该模块将对外提供put、get、delete3个API函数。在我们制定的协议中，发送至Java节点的消息都以元组为基本格式，元组中的元素包括请求标签（put、get或delete）、发送方的pid、对请求的唯一标识符的引用、待查的键，及对应的值（可选，仅用于put）。这套协议简洁明了，用Java处理起来也十分方便。

这些API函数的第一个参数都是Java节点的节点名，同时还假定Java节点的信箱的注册名固定为hbase_server。（与HBase对接的Java节点可以有多个，只要节点名不重复就行；信箱名作为公开的入口位置，硬编码到协议中也没有什么不妥。）由此，不难写出sc_hbase:put/3的实现：

```
put(Node, Key, Value) ->
  Ref = make_ref(),
  {hbase_server, Node} ! {put, self(), Ref, term_to_binary(Key),
                          term_to_binary(Value)},
  receive
    {reply, Ref, ok} ->
      ok
  after 3000 ->
      {error, timeout}
  end.
```

将键、值转换为二进制串

此处之所以用make_ref()为每个请求创建引用，是为了能在receive表达式中过滤出与特定请求相对应的应答，以免被无关消息“迷惑”。此外，在Erlang代码中，键和值都被term_to_binary/1转换成了二进制串；消息元组中各元素数据类型的确定和统一，可以有效简化Java代码中的消息解析逻辑。（在HBase看来，无论键还是数据都只是字节序列而已。）如果Java节点一直没有给出应答，最终将触发超时。

get函数则更为简单。它的职责是从Java节点给出的应答中识别出二进制串格式的值，然后再将它转换成Erlang项式。如果应答中的值并非二进制串而是原子not_found，则get返回{error,not_found}：

```
get(Node, Key) ->
  Ref = make_ref(),
  {hbase_server, Node} ! {get, self(), Ref, term_to_binary(Key)},
```

```
receive
  {reply, Ref, not_found} ->
    {error, not_found};
  {reply, Ref, Binary} ->
    {ok, binary_to_term(Binary)}          <-- 将二进制串转换为项式
after 3000 ->
    {error, timeout}
end.
```

最后是delete，该函数与get类似，但只会返回ok[①]：

```
delete(Node, Key) ->
  Ref = make_ref(),
  {hbase_server, Node} ! {delete, self(), Ref, term_to_binary(Key)},
  receive
  {reply, Ref, ok} ->
    ok
after 3000 ->
    {error, timeout}
end.
```

就这么多了。（别忘了在simple_cache.app文件的模块列表中加入sc_hbase。）上述代码中最关键的便是由各API函数定义的协议，也就是请求元组和应答元组的具体格式。

现在转过头来看看HBase桥接接口中的Java代码，相对而言这部分要更为复杂。首先是HBaseConnector类，它封装了HBase系统的核心交互逻辑。

13.3.2 HBaseConnector类

完成了Erlang API，该来实现对应的Java函数了——也就是图13-4中最右侧的各个部件，完成这些部件之后，集中精力将两端调通便万事大吉了。HBase Java API是一套较为重型的通用数据库API。不过我们只需用它实现上一节定义的put、get、delete3个操作就可以了。这些操作将由HBaseConnector类封装。这样一来，其余代码就不用直接跟HBase打交道了。

> **Java 代码和 C 代码该放在哪儿**
>
> 为了将应用中的 Java 代码和 Erlang 代码隔离开来，通常不把.java 文件放在 src 目录下，而是将之放在另外的 java_src 目录下。（C 代码则放在 c_src 下。）C 和 Java 源码编译后的产物（DLL、可执行文件、.class 文件、.jar 文件等）则应放到 priv 目录下，这样在交付发布镜像时就无须将源码文件囊括在内了。

新建simple_cache/java_src目录，并在目录下创建HBaseConnector.java。由于需要访问HBase Java API，必须在源码文件中导入相关的库。编译、运行Java代码时，不要忘记在-cp参数后给出hbase-<version>.jar文件（位于解压后的HBase目录下）和hadoop-<version>.jar文件（位于解压后的Hadoop Common目录下）的具体位置。源文件的开头如下：

13

① 除非Java节点迟迟不返回应答从而触发超时。——译者注

```
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.*;
import java.util.NavigableMap;
```

紧接着是HBaseConnector类及其构造函数的定义：

```
public class HBaseConnector {
  private HTable table;
  public HBaseConnector() throws Exception {
    super();
    table = new HTable(new HBaseConfiguration(), "cache");      <—  传入配置对象
  }

  // the rest of the code goes here
}
```

简单起见，本章中的Java类不属于任何Java包。编译时将.class文件与源文件一起直接放在java_src目录下即可。

HBaseConnector类只有一个成员变量，这便是用于访问HBase表的HTable对象（由HBase Java API定义）。HTable类的构造函数以HBaseConfiguration对象为参数，具有良好的可定制性。根据我们当前的需求，默认配置即可满足需求，因此只需向构造函数中传入一个全新的配置对象，再附上打算访问的数据库表（即在13.2.2节创建的cache表）的表名就可以了。

完成初始化之后，就轮到get、put和delete方法了。如前文所述，键与二进制值之间的映射关系就存在先前建立的HBase表中。首先是负责从数据库中读取所需值的get方法：

```
public byte[] get(byte[] key) throws Exception {
  // Throws NullPointerException if key is not found
  Result result = table.get(new Get(key));
  NavigableMap<byte[], NavigableMap<byte[], byte[]>> map =
      result.getNoVersionMap();
  return map.get("value".getBytes()).get("".getBytes());
}
```

先构造一个Get对象来描述欲获取的数据条目，再以它为参数调用table对象的get方法，这便是调用HBase API从表中读出所需的值的全过程。值得注意的是，无论是get方法的key参数，还是通过其他方法传给HBase的其他参数，全都是字节数组。在HBase中，一切数据都是字节序列，没有类型之分。在与HBase交互时，一切数据都得转换成字节数组，这也是我们需要在API之外再做一层包装的原因之一。

get方法会返回一个Result对象，借助它便可以访问从数据库中读取到的值。为了访问数据，我们必须先创建一个用于描述HBase中的数据的NavigableMap对象，再将结果读入这个映射。听起来有点儿绕：首先用（字节数组形式的）字段名调用NavigableMap对象的get方法，该方法将返回另一个NavigableMap对象，其中包含按域区分的多个不同的值。后续我们将会提到，在向HBase写入数据时，我们指定的域为空。因此，读取数据时照葫芦画瓢，以空字节数组为参数调用第二个NavigableMap对象的get方法，即可获得与给定的键相对应的值。

put方法更为简单，跟get差不多，只是方向相反：

```
public void put(byte[] key, byte[] value) throws Exception {
  Put put = new Put(key);
  put.add("value".getBytes(), "".getBytes(), value);
  table.put(put);
}
```

首先用给定的键创建一个Put对象。接着，以value为字段名、以空串为域，为Put对象补上与键相对应的值（其中字段名和域都是字节数组）。然后，调用table对象的put方法将数据写入HBase。

3个API中，最简单的便是delete方法：

```
public void delete(byte[] key) throws Exception {
  Delete del = new Delete(key);
  table.delete(del);
}
```

首先用待删除的键创建一个Delete对象，然后用它调用table对象的delete方法即可。

现在你已经掌握了执行基本数据库操作的方法，该轮到Java节点的核心部件了：那就是负责将Java和Erlang对接到一起的HBaseNode类（参见图13-4）。

13.3.3　Java中的消息处理

Java节点的主要框架与13.1.4节中的通信示例差不多。具体来说，它负责接收发自Erlang的请求消息，对请求进行分析和拆解之后再分别进行处理。更有意思的是，我们还用上了异步处理机制，每条消息都会由单独的Java线程负责处理，而不会简简单单地一次只处理一个请求。Java的并发处理能力远不如Erlang，给每个请求单开一个线程的话，效率太低。因此，我们求助于Java标准库提供的线程池。好在标准库为我们化解了大部分多线程相关的复杂性。

通信瓶颈

在当前的系统设计下，有一点需要格外注意。虽然用上了线程池（参见代码清单 13-2），但 OtpMBox 对象仍然只有一个，而它将会成为请求处理的瓶颈。当前这个应用倒还好；要求更为严格的应用则可能无法容忍这个瓶颈。在解决这类问题时，Erlang 中的很多方法也同样适用于 Java 场景下。

和示例代码中一样，我们需要一个类来表示节点的主入口。这个类负责初始化Java节点和用于接收外来请求的信箱。接着，节点进入循环，不断读取信箱中的消息。每处理一条消息，就构造一个任务对象并将之推入队列，由这个对象负责在另外的线程中处理请求并执行数据库操作。请求处理完毕后，该对象会向请求的发起方回传一个应答。全过程中的数据流程和控制流程如图13-5所示。

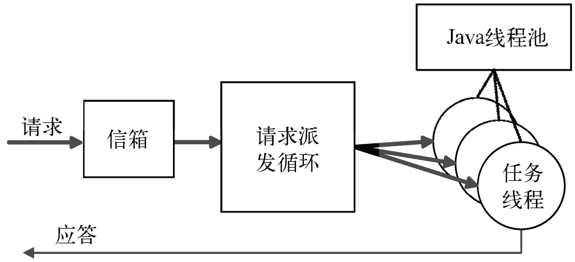


图13-5　在Java线程池的线程中处理请求。这样可以同时处理多个请求，增加系统的吞吐能力

Java节点的主入口是`HBaseNode`类（位于新建的源文件java_src/HBaseNode.java内）。代码清单13-2展示的便是这个类的定义、构造函数，还有`main`方法。不妨和13.1.4节中的`JInterfaceExample`类做个比较。本例中的信箱名是固定的，因此构造函数只有两个参数。

代码清单13-2　`HBaseNode`类

```
import com.ericsson.otp.erlang.*;
import java.util.concurrent.*;                    ❶ 用于引入ExecutorService

public class HBaseNode {
  private HBaseConnector conn;
  private ExecutorService exec;
  private OtpNode node;
  private OtpMbox mbox;

  public HBaseNode(String nodeName, String cookie)
  throws Exception {
    super();
    conn = new HBaseConnector();                  ❷ 创建HBaseConnector实例
    exec = Executors.newFixedThreadPool(10);      ❸ 创建Java线程池
    node = new OtpNode (nodeName, cookie);
    mbox = node.createMbox("hbase_server");
  }

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.println("wrong number of arguments");
      System.out.println("expected: nodeName cookie");
      return;
    }
    HBaseNode main = new HBaseNode(args[0],args[1]);
    main.process();
  }

  // the rest of the code goes here
}
```

除了`JInterfaceExample`构造函数中的铺垫工作以外，此处还新建了一个用于连接本地HBase服务器的`HBaseConnector`对象❷，以及一个用于派发请求的线程池❸。（`ExecutorService`类由Java标准库提供。使用它之前必须先导入java.util.concurrent包中的相关定义❶。）

然后是main()方法，和JInterfaceExample类中的基本上一模一样，只不过不再需要通过命令行参数传入信箱名。和之前一样，程序的主循环也位于process方法中，如代码清单13-3所示。这个无穷循环一直等待着外来消息，跟Erlang/OTP gen_server差不多。新消息到来后会被从信箱的队列中取出、分析，然后派发至新的HBaseTask对象以待处理。（请注意这段代码并不负责向调用方发送应答。）

代码清单13-3　HBaseNode.process()

```
// message format: { Action, FromPID, UniqueRef, Key [, Value] }
private void process() {
  while (true) {
    try {
      OtpErlangObject msg = mbox.receive();                              分解消息 ❶
      OtpErlangTuple t = (OtpErlangTuple) msg;
      String action = ((OtpErlangAtom) t.elementAt(0)).atomValue();
      OtpErlangPid from = (OtpErlangPid) t.elementAt(1);
      OtpErlangRef ref = (OtpErlangRef) t.elementAt(2);
      byte[] key = ((OtpErlangBinary) t.elementAt(3)).binaryValue();
      byte[] value;
      HBaseTask task = null;                                 创建HBaseTask对象 ❷
      if (t.arity() == 5 && action.equals("put")) {
        value = ((OtpErlangBinary) t.elementAt(4)).binaryValue();
        task = new HBaseTask(mbox, conn, from, ref, action, key, value);
      } else if (t.arity() == 4 && action.equals("put")) {
        task = new HBaseTask(mbox, conn, from, ref, action, key, null);
      } else if (t.arity() == 4 && action.equals("delete")) {
        task = new HBaseTask(mbox, conn, from, ref, action, key, null);
      } else {
        System.out.println("invalid request: " + t);
        continue;
      }
      exec.submit(task);
    } catch (Exception e) {                             ❸ 将HBaseTask提交至线程池
        System.out.println("caught error: " + e);
    }
  }
}
```

如果发现消息格式错误，则向标准输出打印错误信息。（请求元组的格式不正确的话，就没法确定发送方是谁，自然也就无法发送应答。）对于需要达到线上产品质量的代码，则应该采用log4j等日志服务；就本例而言，打印到标准输出就够了。

和JInterfaceExample一样，首先对消息进行分解❶，提取出消息中的元素并转换成恰当的类型。过程中如果发生错误，抛出的异常将被捕获并打印出来，代码则继续循环处理队列中的下一条消息。

收到的请求元组必定有 4 或 5 个元素。get或delete消息只有 4 个元素，但put消息有 5 个。在为后者创建HBaseTask对象时，需要赋予一个有效的数据字段；对于前者，将数据字段设置为null即可❷。如果元组的元数既不是 4 也不是 5 ，则打印一条错误信息，然后继续处理下一条消息。

13

在 Java 和 Erlang 中分解消息

请思考一下 Java 等语言中的消息分解方式，再跟 Erlang 中的模式匹配做一个对比。在 Java 中，分解这个小小的元组并决定后续的动作足足花了 10 行代码。如果用 Erlang 实现，只需要这样：

```
case Message of
  {Action, From, Ref, Key} -> ...;
  {Action, From, Ref, Key, Value} -> ...;
end.
```

一个冗长繁琐，一个简洁明了，高下立判。倒不是要故意刁难 Java，只是为了衬托一下声明式编程的强大。简洁的代码一目了然，因此 bug 也更难容身。

最后，创建完持有数据的HBaseTask对象之后，将之投递到线程池中❸以便进行异步请求处理，同时主循环继续处理后续的消息。HBaseTask是图13-4中的最后一个有待实现的组件，不过并不复杂。

13.3.4　HBaseTask类

有意思的事情都集中在HBaseTask类中，如代码清单13-4所示。由于实现了Java标准库中的Runnable接口，这个类以run方法为主入口。在本例中，run只负责分析请求中的动作并将之派发给处理相应操作的方法。（至此，从Erlang节点处发来的请求元组中的各个字段已经被分解成了更易于使用的形式。）get、put、delete请求分别由doGet、doPut、doDelete方法处理。

代码清单13-4　HBaseTask类

```
import com.ericsson.otp.erlang.*;

public class HBaseTask implements Runnable {
  private OtpMbox mbox;
  private HBaseConnector conn;
  private OtpErlangPid from;
  private OtpErlangRef ref;
  private String action;
  private byte[] key;
  private byte[] value;

  public HBaseTask(OtpMbox mbox, HBaseConnector conn,
                   OtpErlangPid from, OtpErlangRef ref,
                   String action, byte[] key, byte[] value) {
    super();
    this.mbox = mbox;
    this.conn = conn;
    this.from = from;
    this.ref = ref;
    this.action = action;
    this.key = key;
    this.value = value;
  }
```

```
    public void run() {
      try {
        if (action.equals("get")) {
          doGet();
        } else if (action.equals("put")) {
          doPut();
        } else if (action.equals("delete")) {
          doDelete();
        } else {
          System.out.println("invalid action: " + action);
        }
      } catch (Exception e) {
          System.out.println("caught error: " + e);
      }
    }

    // the rest of the code goes here
  }
```

doGet、doPut和doDelete方法负责执行具体操作。run方法将会捕获这些方法抛出的所有异常并在向Erlang回传的应答中报告错误。首先来看下doGet：

```
  private void doGet() throws Exception {
    OtpErlangObject result;
    try {
      result = new OtpErlangBinary(conn.get(key));
    } catch (NullPointerException e) {
      result = new OtpErlangAtom("not_found");
    }
    OtpErlangTuple reply = new OtpErlangTuple(new OtpErlangObject[] {
                                 new OtpErlangAtom("reply"), ref,
                                 result
                               });
    mbox.send(from, reply);
  }
```

给定一个键，该方法便会读出对应的二进制串格式的值（如果找不到，就发回原子not_found）。请注意，经过HBase API的抽象，利用HBaseConnection conn读取值的过程大大简化了。接下来，是doPut方法：

```
  private void doPut() throws Exception {
    conn.put(key, value);
    OtpErlangTuple reply = new OtpErlangTuple(new OtpErlangObject[] {
                                 new OtpErlangAtom("reply"), ref,
                                 new OtpErlangAtom("ok")
                               });
    mbox.send(from, reply);
  }
```

doPut跟doGet如出一辙，负责向HBase中插入与指定的键相对应的值。本例中该方法仅向Erlang方面返回一个"ok"。最后，doDelete也差不多：

```
  private void doDelete() throws Exception {
    conn.delete(key);
    OtpErlangTuple reply = new OtpErlangTuple(new OtpErlangObject[] {
```

```
                                new OtpErlangAtom("reply"), ref,
                                new OtpErlangAtom("ok")
                              });
    mbox.send(from, reply);
  }
```

好了，Java这侧便告一段落了。至此，我们有了一个可运行的Java节点，该节点提供了一套支持HBase上的get、put、delete操作的接口。现在只剩下跟Simple Cache应用整合了。

13.4　在 Simple Cache 中整合 HBase

要想在Simple Cache应用中用上新鲜出炉的Erlang-HBase接口，就必须改写lookup、insert和delete 3 个函数，让它们像本章开头处所描述的那样与HBase交互，从而将缓存数据存入HBase表。好在我们早先为缓存应用写下的代码布局合理，现在只需修改图13-6中的前端模块simple_cache.erl就可以了。除了在第7章中为了支持日志功能而添加的sc_event调用以外，该模块目前的内容跟代码清单6-7基本一致（参见6.4.3节）。

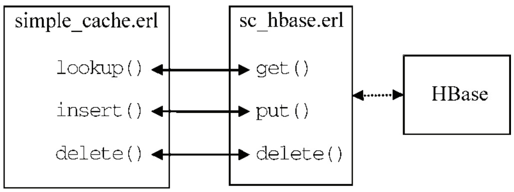


图13-6　在simple_cache应用中集成sc_hbase.erl模块。为了与HBase交互，需要改写lookup、insert和delete 3 个函数

为了方便日后修改，我们首先用HBASE_NODE宏来指代HBase Java节点的节点名。请将以下代码添加到文件顶部的导出声明下方：

```
-define(HBASE_NODE, 'hbase@localhost').
```

（按理说，我们应该将节点名设计成simple_cache应用的一个配置项。这个改动就留给你当做练习吧。）下面我们开始依次修改各个API函数。除lookup函数以外，其余两个函数只需修改一行代码。

13.4.1　查询

首先在simple_cache:lookup/1加入HBase查询逻辑，当（且仅当）在本地节点上找不到待查的数据条目时，尝试从HBase中查询数据。新版本如代码清单13-5所示。和此前一样，我们利用try/catch来捕获函数执行过程中的各种错误（包括fetch和get调用的返回值与{ok, ...}模式不匹配的情况），并对外返回not_found。

代码清单13-5　新版的simple_cache:lookup/1

```
lookup(Key) ->
    sc_event:lookup(Key),
```

```
    try
        case sc_store:lookup(Key) of                    尝试本地查询
            {ok, Pid} ->
                {ok, Value} = sc_element:fetch(Pid),
                {ok, Value};
            {error, _} ->                               尝试从HBase中获取数据
                {ok, Value} = sc_hbase:get(?HBASE_NODE, Key),
                insert(Key, Value),
                {ok, Value}
        end
    catch
        _Class:_Exception ->
            {error, not_found}
    end.
```

本地节点上找不到，就到HBase中去找。找到之后，将读出的数据条目就地插入缓存，用以加速后续的查询速度。如果HBase中也找不到，返回值必不匹配`{ok, Value}`模式，控制权随即落入`try`表达式的手中。

最麻烦的部分也不过如此了；相较之下，`insert`和`delete`函数的改动完全是小菜一碟。

13.4.2 插入

`simple_cache:insert/2`负责将插入缓存的数据原封不动地存入HBase。新版本如下：

```
insert(Key, Value) ->
    case sc_store:lookup(Key) of
    {ok, Pid} ->
        sc_event:replace(Key, Value),
    sc_element:replace(Pid, Value);
{error, _Reason} ->
    {ok, Pid} = sc_element:create(Value),
    sc_store:insert(Key, Pid),
    sc_event:create(Key, Value)
end,                                            同时插入HBase
sc_hbase:put(?HBASE_NODE, Key, Value).
```

只此一行，在新值插入系统时顺便调用一下`sc_hbase:put/2`即可。不管给定的键是否已经存在，HBase的数据写入接口都一视同仁[①]，因此一行代码就能搞定。不过，为了避免构成竞态条件，应该先写缓存再写HBase，否则与插入操作同时进行的查询操作有可能会碰到HBase中有数据而缓存中无数据的情况[②]。

最后，针对`delete`函数的改动跟`insert`相差无几。

13.4.3 删除

从缓存中删除数据时，必须同时清理HBase表。新版本的`simple_cache:delete/1`如下：

13

① 也就是说，HBase的put接口具有幂等性。——译者注

② 这会导致`lookup/1`重复调用`insert/2`。——译者注

```
delete(Key) ->
    sc_event:delete(Key),
    case sc_store:lookup(Key) of
    {ok, Pid} ->
        sc_hbase:delete(?HBASE_NODE, Key),     <-- 先清除HBase中的数据
        sc_element:delete(Pid);                <-- 再清除缓存中的数据
    {error, _Reason} ->
        ok
    end.
```

请注意，sc_hbase:delete/2调用必须先于sc_element:delete/1调用，以防与删除操作并发执行的查询操作将本该删除的值重新插入缓存。如果先清理缓存，新的查询请求可能会在HBase中的数据删除完毕之前重新将数据读出并插入缓存。

至此，上一节中的3个Java类（HBaseConnector、HBaseNode和HBaseTask）应该也已经编译完毕了吧，现在将各个环节串接起来跑跑看吧。

13.5　运行集成系统

我们的缓存系统已经比较复杂了，要想让它跑起来，必须依次启动以下服务。

- 启动HBase（参见13.2.1节）。
- 启动至少一个联络节点（为了给后续启动的Java节点准备EPMD，请至少在本地机器上放一个）。例如：erl -sname contact1 -detached -setcookie secret。
- 启动HBase桥接接口节点，注意cookie应与Erlang节点保持一致。详情请见下文。
- 启动simple_cache系统（参见10.2.6节）。例如：erl -sname mynode -pa ./simple_cache/ebin -pa ./ resource_discovery/ebin -boot ./simple_cache -config ./sys -setcookie secret。
- 如果节点相互不可见，请检查各个节点是否都采用-sname参数启动（此处假定HBase Java节点采用的也是短节点名——简而言之就是节点名的主机部分不含“.”），并确认所有节点的cookie完全一致。

HBase Java节点的启动方式与13.1.5节中的示例差不多，不过要想运行HBase桥接接口，还得用上几个Java库。除了编译时用到的OTP Jinterface库、HBase库和Hadoop Common库（参见13.2.1节）以外，还需要：

- Apache Commons Logging，参见http://commons.apache.org/logging/；
- log4j，参见http://logging.apache.org/log4j/；
- Apache ZooKeeper，参见http://hadoop.apache.org/zookeeper/。

好在这些库的JAR包在HBase的lib目录下都有，为你免除了逐一安装的麻烦。

当然，为了定位HBase桥接接口的.class文件，还得在Java的class path中加上simple_cache/java_src目录；如果选择将生成的.class文件与.java源码文件隔离开来，那么就应该是simple_cache/priv/java目录。启动Java节点的命令很繁琐，大概是这样（以下内容全部位于同一行，只因篇幅所限才做了折行处理）：

```
java -cp simple_cache/java_src:/path/to/OtpErlang.jar:/path/to/
➥ hbase-<version>.jar:/path/to/hadoop-<version>-core.jar:/path/to/
➥ commons-logging-<version>.jar:/path/to/log4j-<version>.jar:/path/to/
➥ zookeeper-<version>.jar   HBaseNode hbase secret
```

其中hbase是节点名，应与simple_cache.erl中定义的HBASE_NODE宏保持一致（13.4节）；secret是Java节点的cookie字符串。可以通过设置Java专用的CLASSPATH环境变量来取代-cp命令行参数。此外，还可以写个shell脚本来执行该命令。

一切顺利的话，HBase数据库、Erlang-HBase桥接接口节点、simple_cache目标系统就悉数启动完毕了。现在，可以通过simple_cache模块完成缓存数据的插入和查询，还能直接用sc_hbase模块查看和操纵HBase中的内容，正如以下交互会话过程所示：

```
Eshell V5.7.3  (abort with ^G)
(mynode@localhost)1> sc_hbase:get(hbase@localhost, foo).
{error,not_found}
(mynode@localhost)2> simple_cache:insert(foo, bar).
=INFO REPORT==== 24-Apr-2010::21:25:27 ===
create(foo, bar)
ok
(mynode@localhost)3> simple_cache:lookup(foo).
=INFO REPORT==== 24-Apr-2010::21:25:50 ===
lookup(foo)
{ok,bar}
(mynode@localhost)4> sc_hbase:get(hbase@localhost, foo).
{ok,bar}
(mynode@localhost)5> simple_cache:lookup(17).
=INFO REPORT==== 24-Apr-2010::21:27:17 ===
lookup(17)
{error,not_found}
(mynode@localhost)6> sc_hbase:put(hbase@localhost, 17, 42).
ok
(mynode@localhost)7> sc_hbase:get(hbase@localhost, 17).
{ok,42}
(mynode@localhost)8> simple_cache:lookup(17).
=INFO REPORT==== 24-Apr-2010::21:29:09 ===
lookup(17)
=INFO REPORT==== 24-Apr-2010::21:29:09 ===        ❶ 自动缓存HBase中的数据
create(17, 42)
{ok,42}
(mynode@localhost)9> simple_cache:lookup(17).      ❷ 缓存查询直接命中
=INFO REPORT==== 24-Apr-2010::21:34:49 ===
lookup(17)
{ok,42}
(mynode@localhost)10> simple_cache:delete(foo).
=INFO REPORT==== 24-Apr-2010::21:29:41 ===
delete(foo)
ok
(mynode@localhost)11> simple_cache:lookup(foo).
=INFO REPORT==== 24-Apr-2010::21:29:44 ===
lookup(foo)
{error,not_found}
(mynode@localhost)12> sc_hbase:get(hbase@localhost, foo).
{error,not_found}
```

我们来看看以上交互过程中都发生了些什么。首先，我们查得HBase中不存在foo。将foo插入缓存后，从缓存和HBase中都可以查得foo。

接着，我们查得缓存中没有17。将17→42直接写入HBase，再做一次查询，缓存会从HBase中读出17并将17→42插入缓存。在接下来的查询中，17直接命中缓存。最后，从缓存中删除foo，经验证HBase中的foo也被同时删除。

如你所见，一切都运转自如。有一条info日志消息create(17,42)❶特别值得注意：缓存中找不到17时，向HBase发起查询（必须事先向其中插入），并将读出的值自动存入缓存以备后用。下一次查询果然直接命中缓存❷。用户想必会欣喜万分。

13.6　小结

现在你已经掌握了利用Jinterface和Java创建非Erlang节点并与之交互，进而将之集成到系统中去的方法。同样的手段还可用于创建任意Java库的桥接接口。

本书的旅程就快结束了。但愿你所习得的知识对得起这一路上的坎坷：Erlang语言；OTP应用中的行为模式、监督机制、日志，还有事件处理；分布式Erlang、Mnesia以及发布镜像的制作；通过HTTP、端口、驱动以及Jinterface集成外围代码。在下一章中，我们将学习代码测评和性能优化相关的技术，从而让你的应用跑起来更加顺畅。

第 14 章

优化与性能

14

没有绝对的快，只有够不够快的问题。
——Joe Armstrong

本章概要

- ❑ 如何进行性能调优
- ❑ 利用cprof和fprof分析代码
- ❑ Erlang编程语言的局限和陷阱

除非到了必须为那几毫秒、几千字节锱铢必较的地步，否则不要总想着去做优化。现在，Erlware团队便已经“落入”了这种境地。事实表明，很多人都希望能有一个集中式的Erlang软件下载站。经过速度提升和功能改善，同时也拜你开发的简单缓存服务所赐，erlware.org的访问量日益增长。效率问题又开始凸显。当前我们没法增加硬件投入，要想靠添加机器来实现水平扩展那是不可能了。Erlware团队必须优化代码，尽可能“榨取”每一个时钟周期。

Martin 说……

我还记得 2002 年在美国的函数式编程会议上头一回碰到 Joe Armstrong 时的情景。他是 Erlang 的创始人之一，满口金玉良言和各种计算机科学方面的箴言警句；这些话有些出自他本人，有些则源自别处。有些话深深地吸引了我，其中有一句我特别喜欢：“先跑起来，再去追求美；最后，只有实在没有别的办法的时候，才去追求速度。”他接着说，“然而在 90%的情况下，速度与美同在。因此本质上讲，全心全意地追求美就可以了！”

你的代码已经足够精美了，我们只能从底层开始优化。本章的目的便是向你介绍一些人的用于优化先前章节中编写的代码工具，从而为Erlware团队提供帮助。在正式进入这一章之前，我们希望你能明白，以简洁、可读性和可维护性为代价去换取性能并非上选；只有在代码已经打磨得精美绝伦却仍然无法满足需求时，方能出此下策。本章介绍的正是那些极少数无法通过追求代码之美来达成目标的情况。

性能优化唯一的秘诀便是系统化。有些问题非常明显，一眼就能看出来；除了这些问题以外，你都应当先行度量，设定好基准线并寻找瓶颈，优化之后再次度量以检验性能是否得到改善。我

们将在本章探讨相关的工具和技巧。

首先，我们将以系统化的方式讨论如何进行性能调优，如何找出问题所在，以及如何确认问题修复与否。

14.1　如何进行性能调优

从许多方面来看，性能调优算得上是一门艺术。有些人在探寻和消除瓶颈方面尤为擅长，但那多半出于多年经验结晶而成的直觉——这些东西是难以从书中习得的。本章将从方法论的角度诠释性能调优这门科学。

Erlware团队打算按照图14-1中的步骤系统化地进行性能调优。我们来仔细考察一下这些步骤。

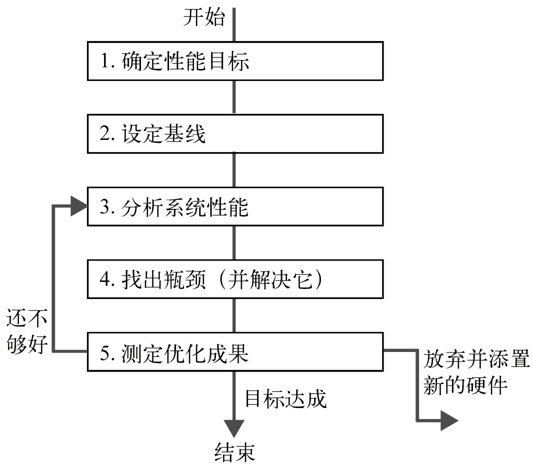


图14-1　性能调优过程：确定目标、设定基线、分析性能、优化、再度分析。循环往复直至达成目标或放弃

14.1.1　设定性能目标

在开始调优之前，应该先明确怎样才算达标。目标设定应满足以下几个特征（亦被称作SMART原则）。

- 具体（Specific）——表意清晰，例如改善CPU使用率或每秒的吞吐量。
- 可度量（Measurable）——可通过系统化的度量进行验证。
- 可达成（Attainable）——可以切实达成。能在整个项目力所能及的范围内实现目标；否则便是好高骛远。
- 现实（Realistic）——结合当前的资源和动因，如果必须靠整个团队竭尽全力方可达成，那多半不现实。
- 有时限（Timely）——可在指定时限内完成。性能优化不能没完没了：设定时限有助于集中精力，避免无休止地纠缠于细枝末节。

例如，Erlware团队将根据erlware.org的网站点击量以及系统中各个部分的事务量来确定本次性能优化的目标。他们还打算从过往几个月内的统计数据中得出总体的流量增长趋势，然后综合公司当前的推广力度，对未来6个月内的流量作出预测。这些便是团队树立性能调优目标的依据所在。

14.1.2 设定基线

如果目标已经可以量化（本应如此），便可以设立基线了。按照先前立下的标杆，通过测试找出自己与目标之间的差距。比如，系统当前的CPU消耗和吞吐量分别是多少（说的是调优之前的情况），基线的覆盖面越广，就越容易测定代码改动的影响。

14.1.3 系统性能分析

如果实测表明目标尚未达成——有时候你会发现自己只是杞人忧天，或者至少实际问题并非之前想象的那样，这时你就该好好考虑一下了，时间（或钱、带宽）都花到哪里去了？代码性能分析可以帮你揪出很多问题，譬如CPU瓶颈、内存消耗大户、锁的争抢热点，或是因I/O而阻塞的进程等。

14.1.4 确定需要解决的问题

确定了主要问题之后，必须决定应该立即着手解决的问题。纵观全局，最重要的问题有可能会因为成本过高（且难以确定收益）而不得不延后。有些问题则可能相对简单，更易于解决，而且无须对核心组件大动干戈便能达成当下的优化目标。那些收效显著，且能够在既定的时间内完成的问题才是最佳选择。另外不要一次性修复所有问题：一个一个来，这样你才能确认哪些改动会对性能产生直接的影响。

14.1.5 测定优化成果

修改代码之后再次进行测定，并将结果与之前设定的基线进行比较。新的代码是否有所改善？（等你看到经过“明显改善”的代码被屡屡测定为毫无帮助甚至起了反作用时，可不要太过惊讶。）如果确有改善，是否已经达到先前设定的性能目标？达到了的话，那就恭喜了——这一轮调优暂时宣告结束。否则，折回图14-1中的步骤3并持续进行性能分析、修改、测定，直至达成目标（或者因时间消耗殆尽而不得不掏腰包更新硬件）为止。

可用于系统性能测定的工具很多，从简单的日志等代码监测手段，到etop、percept以及开源的eper等各种Erlang专用工具，再到更外层的操作系统级的专用工具等，不一而足。其中大部分都超出了本书的范畴。在本章中，我们将为你介绍Erlang/OTP标准库中自带的Erlang代码性能分析工具。详情请参见下一节。

14.2　Erlang 代码性能分析

要想抓住系统中的性能瓶颈，最可靠的方法就是性能分析。实践多了，你就会练就一双火眼金睛，可以一眼看出隐藏在代码中的性能问题；但性能瓶颈的定位迟早还是要靠性能分析。

一般来说，性能分析就是边运行代码边采集统计数据，然后将采集到的性能数据与代码中的位置进行配对的过程。最典型的性能测试便是测定代码中耗时最长的部分：它们就是主要瓶颈之所在。测定耗时的方法多种多样：CPU耗时用于描述程序的实际负载，物理耗时（wall-clock time）则用于描述程序运行的总时长。物理耗时长而CPU耗时短，表示代码大部分时间都处在等待状态（一般是磁盘或网络I/O）。CPU耗时过高，则表示算法太烂（例如平方复杂度或指数复杂度的算法）。一些简单的性能分析器难以精确记录CPU时间，而用每个函数的调用次数或每行代码的执行次数来近似表示CPU时间。性能分析器还可以度量代码中不同位置上的内存消耗、I/O消耗、处于就绪状态的进程数等。（程序中处于就绪状态的进程过少，往往意味着碰上了同步瓶颈，从而导致系统并发度降低。）

Erlang/OTP自带一系列性能分析工具，包括用于检查代码覆盖率的cover、用于分析内存使用量的instrument模块，以及最近出现的用于进行并发性能分析的percept等工具。在这一节中，我将介绍Erlang/OTP中的两大时间分析工具：fprof和cprof。首先是用法较为简单的cprof。

14.2.1　用cprof计算调用次数

cprof工具输出的信息不如fprof丰富，但用法却十分简单。cprof用于记录函数调用次数。相对于fprof，它的主要优势在于对运行时系统的影响很小（被分析的代码大约会慢上10%）。因此，如果要对线上运行着的代码进行性能分析，使用cprof更为合适。

代码清单14-1中的示例模块profile_ex除了空跑一小段时间以外什么也不干。这段代码的唯一目的就是用做性能分析的对象。（缓存应用不适合拿来用做性能分析示例，它过于庞大，而且也不是CPU密集型的代码。）

代码清单14-1　用做性能分析对象的示例代码

```
-module(profile_ex).

%% API
-export([run/0]).

run() ->
    spawn(fun() -> looper(1000) end),
    spawn (fun() -> funner(1000) end).

looper(0) ->
    ok;
looper(N) ->
    _ = integer_to_list(N),
    looper(N - 1).

funner(N) ->
```

```
        funner(fun(X) -> integer_to_list(X) end, N).

    funner(_Fun, 0) ->
        ok;
    funner(Fun, N) ->
        Fun(N),
        funner(Fun, N - 1).
```

`run()`函数派生出两个并行进程：第一个进程运行`looper/1`函数，第二个进程运行`funner/1`函数，两个函数的参数值都是1000。`looper/1`函数从1一直循环到输入参数给定的值，依次调用`integer_to_list(N)`并扔掉求值结果。`funner/1`函数的功能差不多，但它多用了一个fun函数，并把它用做`funner/2`的第二个参数。

以下Erlang shell会话演示了如何用cprof来测评上述示例模块的性能。

```
Eshell V5.7.4  (abort with ^G)
1> c(profile_ex).
{ok,profile_ex}
2> cprof:start().                                    ❶ 启动 cprof
5359
3> profile_ex:run().
<0.43.0>
4> cprof:pause().                                    ❷ 终止测评
5380
5> cprof:analyse(profile_ex).                        ❸ 获取结果
{profile_ex,3006,
          [{{profile_ex,looper,1},1001},
           {{profile_ex,funner,2},1001},
           {{profile_ex,'-funner/1-fun-0-',1},1000},
           {{profile_ex,run,0},1},
           {{profile_ex,module_info,0},1},
           {{profile_ex,funner,1},1},
           {{profile_ex,'-run/0-fun-1-',0},1},
           {{profile_ex,'-run/0-fun-0-',0},1}]}
6> cprof:stop().
5380
7>
```

编译并加载profile_ex模块（`c(...)`会在完成编译的同时加载模块）后，调用`cprof:start()`❶。从此往后，所有模块的函数调用都将被计数。被测代码本身无须做任何修改——无须特殊的编译过程也无须重新加载，编译代码时也无须包含调试信息。cprof适用于任何系统，而且不会对运行中的应用产生干扰，因此特别有用。

很多时候，我们无须关注整个系统中的所有模块，只对特定模块中的函数调用次数感兴趣。例如，通过`cprof:start(profile_ex)`调用可以单独测试profile_ex模块的性能。而这又会进一步降低调用次数统计对运行中的系统的影响。不过，目前为止性能测试带来的开销还不成问题。

启动cprof并告知待测模块，然后启动待测代码。测评完成后，令cprof暂停，停止对函数调用进行计数❷。只需调用`cprof:analyse(profile_ex)`，便可获取profile_ex模块中所有函数的调用次数。

返回的项式很容易解读：函数按调用次数降序排列。可以看出，结果与预期一致：主循环函

数looper/1和funner/2各被调用1001次。（在第1001次调用时计数器归零。）其中还有一个命名古怪的函数，名为`'-funner/1-fun-0-/1'`，调用次数刚好是1000次：该函数由编译器生成，代表funner/1函数中的fun函数。（循环计数器归零时该函数未被调用。）其他函数各被调用一次。注意到此处并未显示`integer_to_list/1`调用。首先，这个内置函数（BIF）并不属于profile_ex模块；其次，即便试图单独针对erlang模块进行评测，cprof也无法记录Erlang BIF的调用次数。

cprof可以帮助我们快速确定哪些函数被调用，以及各被调用了多少次。有时候掌握了这些信息就足够了，但通常我们还需要更加详细的信息。这时便可以借助于fprof工具。

14.2.2　用fprof测定执行时间

在各种标准性能测试工具之中，最有价值的大概就属fprof了。它采集的信息内容翔实、格式清晰，而且十分易于使用，无疑是诊断性能问题的首选工具。fprof设计用于取代旧的eprof（在Erlang/OTP发行版中仍然能找到eprof，但相较之下效率却低得多）。fprof和eprof都以Erlang的跟踪机制为基础，因此既不需要对代码做特殊编译，也不需要保留调试信息。它们的运行时开销比cprof要大得多，会将代码的运行速度拖慢10倍以上，用于线上系统时应格外小心。

不可或缺的 runtime_tools 应用

fprof 依赖于 runtime_tools 应用中的 dbg（详情参见 Erlang/OTP 文档中的工具章节）。使用 fprof 前必须确保系统中装有该应用。标准 Erlang/OTP 发行版内包含 runtime_tools，但如果用操作系统自带的软件包管理器来安装 Erlang，那么最好检查一下是否安装了这个包。fprof 模块，连带 cprof、cover、instrument 以及其他一些工具，都属于 tools 应用。

fprof的调用方法和cprof类似。以下会话展示了如何用fprof来分析上一节中的profile_ex模块：

```
Eshell V5.7.4  (abort with ^G)
1> c(profile_ex).
{ok,profile_ex}
2> fprof:trace(start).                    ❶ 开始跟踪
ok
3> profile_ex:run().
<0.42.0>
4> fprof:trace(stop).
ok
5> fprof:profile().                                  ❷ 处理跟踪过程
Reading trace data...
.............
End of trace!
ok
6> fprof:analyse({dest, "profile.txt"}).          ❸ 分析性能分析数据
Processing data...
Creating output...
Done!
ok
7>
```

fprof并没有在shell中输出可供直接查看的项式，而是在当前目录下新建了一个名为profile.txt的文件。第一步是让fprof开始跟踪❶。跟踪信息默认写入名为fprof.trace的文件，该文件是二进制格式，不适合直接阅读。第二步，像之前一样调用`profile_ex:run()`，稍等一会儿，然后停止跟踪。接着，调用`fprof:profile()`❷从而将跟踪信息转换为供性能评测用的原始数据。转换后的数据由fprof服务器保存于内存中。然后，调用`fprof:analyse/1`❸分析数据，将分析结果输出至指定的文件（本例中是profile.txt）。

启动跟踪并得到最终输出的方法多种多样，我们将之留作练习，请研读fprof的文档找出答案。待评测代码的规模也很重要。和cprof不同，fprof会迅速积攒数据，持续对大段代码进行评测的话，fprof会吐出好几吉比特的评测信息。

能得出数据不等于会解读数据，这便是我们接下来的主题。

诠释fprof的输出

分析结果全部位于profile.txt文件内。文本内容采用的是Erlang项式的格式（以句点结尾），时而夹杂一些Erlang注释（以`%`开头），在跟Erlang打交道的过程中，这可谓是司空见惯了。这样一来，你便可以轻而易举地利用标准库函数file:consult（Filename）读入数据并进行各种统计。粗看上去，这种格式比较复杂，但只要掌握了诀窍，理解起来也不算难。我们将逐一诠释文件中各个重要部分的含义。

文件的开头是注释`Analysis results`（分析结果），紧跟的是分析启动时给定的选项（有助于后续重跑分析数据）：

```
%% Analysis results:
{  analysis_options,
 [{callers, true},
  {sort, acc},
  {totals, false},
  {details, true}]}.
```

紧跟着的一段包含函数的总调用次数（`CNT`）、完整执行过程的总耗时（`ACC`，以毫秒为单位），以及文件中列出的所有函数的总耗时（`OWN`）。

```
%                                               CNT       ACC       OWN
[{ totals,                                     5045,   78.976,   78.929}].
```

换言之，总的`OWN`耗时中并未涵盖未测评函数所消耗的时间。在本例中，我们并未将测评过程限定在某个特定模块之内，因此`OWN`和`ACC`很接近。一般而言，固有时长（own time）指单个函数自身耗费的时长，其中不包括它所调用的其他函数的耗时；而累计时长记录的则是从头到尾的总时长。度量时间时，默认采用的是物理时长，通过`cpu_time`选项可以对此进行调整。

紧跟着总耗时信息之后，分段列出了跟踪过程中所有Erlang进程，每段的开头是进程的摘要信息，如下所示：

```
%                                               CNT       ACC       OWN
[{ "<0.51.0>",                                 3012,undefined,   48.973},
 { spawned_by, "<0.38.0>"},
 { spawned_as, {erlang,apply,["#Fun<profile_ex.1.108254554>",[]]}},
 { initial_calls, [{erlang,apply,2},{profile_ex,'-run/0-fun-1-',0}]}].
```

从initial_calls和spawned_as项可以看出，这是由profile_ex:run()派生的两个进程之一的性能测试数据摘要。可以看到这个进程的OWN时长占整个文件的OWN时长的一半还多，这个数据是解释得通的：虽然profile_ex:run()启动的两个进程干的事情都差不多，但在执行策略上一个要比另一个略为复杂一些。单个进程的ACC总是undefined。

顺着文件内容接着往下看，后面还有一个包含类似的initial_calls和spawned_as项的进程数据摘要：

```
%                                                  CNT        ACC        OWN
[{ "<0.50.0>",                                   2011,undefined,   29.338}
 { spawned_by, "<0.38.0>"},
 { spawned_as, {erlang,apply,["#Fun<profile_ex.0.133762870>",[]]}},
 { initial_calls, [{erlang,apply,2},{profile_ex,'-run/0-fun-0-',0}]}].
```

显然这就是profile_ex:run()派生的另外一个进程了。从spawned_by可以看出这两个进程的父进程相同。（从进程标识符<0.50.0>还可以看出，虽然这个进程出现的次序靠后，但它的派生时间比另一个进程要来得早。）分别累加两个进程的CNT列和OWN列，结果与文件开头处的总和大致相等。

进程数据摘要之后（至下一个进程摘要之前）是该进程调用的函数，每个“段落”（对应于一个Erlang项式）一个函数。譬如：

```
{[{{profile_ex,funner,1},                      1,   49.047,    0.025},
  {{profile_ex,funner,2},                   1000,    0.000,   20.692}],
 { {profile_ex,funner,2},                   1001,   49.047,   20.717},     %
 [{{profile_ex,'-funner/1-fun-0-',1},     1000,   28.217,   18.482},
{suspend,                                1,    0.113,    0.000},
{{profile_ex,funner,2},                1000,    0.000,   20.692}]}.
```

每个段落中都有一行末尾标有%——那一行指代的是该段落所关注的函数：在这里，这个函数就是profile_ex:funner/2。它总共被调用了1001次，共耗时49.047毫秒，其中它自身占用了20.717毫秒。在%标记以上，每行代表该函数的一个调用方，其中funner/1调用了1次，funner/2调用了1000次。对照代码清单14-1，评测数据与代码逻辑完全相符。

在%标记以下，每行代表一个被该函数调用的函数。可以看到它不光调用了自己1000次，还调用了'-funner/1-fun-0'/1 1000次。这就是funner进程在总调用次数上比looper进程多出1000次调用的原因。

最后，还可以看出进程在该函数上挂起了一次，历时0.113毫秒。进程挂起也被当做函数调用写入本文件，甚至还独占一个段落：

```
{[{{erlang,apply,2},                          1,   29.427,    0.000},
  {{profile_ex,funner,2},                       1,    0.113,    0.000}],
 { suspend,                                   2,   29.540,    0.000},     %
 [ ]}.                                                                    .
```

代表进程挂起的那一行由%标出。这一段指出该进程分别在erlang:apply/2和funner/2上挂起过一次，共计29.540毫秒。垃圾回收用时也以相同的方式列出：

```
{[{{profile_ex,'-funner/1-fun-0-',1},        6,    0.357,    0.357}],
 { garbage_collect,                          6,    0.357,    0.357},     %
 [ ]}.
```

由此可知该进程执行了6次垃圾回收，共耗时0.357毫秒。

学会诠释这些文件的方法之后，你便会发现它们一点儿也不神秘，非常简单。然而在面对特定应用时，要想清晰诠释评测数据中每个数字的含义，就完全是另外一回事了。

不妨将进程挂起次数和垃圾回收次数作为性能评测数据分析的切入点。当进程等待消息，或调度器将某个进程暂停以方便其他进程运行时，进程便会挂起。所谓垃圾回收，就是运行时系统追踪并回收先前由进程分配但现在已经不再使用的内存的过程。垃圾回收器还负责按需增减进程的堆。执行大量I/O操作或总等着接收消息的进程会时常挂起。分配大量临时数据的进程则会在垃圾回收上耗费更多的时间。优先观察挂起和垃圾回收情况有助于快速判断进程间的耗时差异是否由这两个因素导致。

fprof 文件中的陷阱

跟踪记录文件中充斥着大量数字，解读起来让人头晕目眩。有时候简单的累加会让你误入歧途，如以下两种情况。

- 在复杂的相互递归[①]操作中，函数的 ACC 时长会不准。
- 在测定物理耗时时，操作系统的调度机制会干扰执行时间。碰到这种情况的时候，你会发现某个函数明明没干什么事情，耗时却不短。要是觉得不对头，那么最好重新再来一次，好做个比较。

不要锱铢必较地盘算每一毫秒的去向。将整个文件视作一个整体，这样才能快速定位耗时最多的位置。

对比looper进程和funner进程的挂起时间，你会发现两者基本相同（分别是32.649和29.540毫秒）：

```
{[{{profile_ex,looper,1},                    1,   32.224,    0.000},
  {{erlang,apply,2},                       1,    0.425,    0.000}],
 { suspend,                                2,   32.649,    0.000},     %
 [ ]}.
```

再对比一下垃圾回收时间，你会发现两者都很小（分别是0.164和0.357毫秒）。每次测量，这两个数值间的差值都会波动，但无论如何都只占总执行时间的一小部分：

```
{[{{profile_ex,looper,1},                    6,    0.164,    0.164}],
 { garbage_collect,                        6,    0.164,    0.164},     %
 [ ]}.
```

抛开这些，现在来看看每个进程都调用了哪些函数。我们曾在描述`funner/2`函数的段落中看到该函数累计耗时20.717毫秒；而它又调用了函数`'-funner/1-fun-0-'/1`，文件中描述后者的段落如下：

```
{[{{profile_ex,funner,2},                 1000,   28.217,   18.482}],
 { {profile_ex,'-funner/1-fun-0-',1},     1000,   28.217,   18.482},     %
 [{{erlang,integer_to_list,1},          1000,    9.378,    9.378},
  {garbage_collect,                        6,    0.357,    0.357}]}.
```

① mutually recursive，指A调用B，B又反过来调用A进而形成递归的情况。——译者注

funner/1中的fun表达式共耗时18.482毫秒，在这个fun函数中，integer_to_list/1又耗费了9.378毫秒。几个数值加起来大约是48.5毫秒，跟funner进程总的OWN时长（48.973毫秒）相近。这下这个进程的时间花在哪里就一目了然了。

再看看looper/1函数对应的段落，可以看到它的OWN时长是19.649毫秒，其中它直接调用的integer_to_list/1耗时9.505毫秒（跟funner进程花在同一个函数上的时间相当）：

```
{[{{profile_ex,'-run/0-fun-0-',0},          1,   61.542,    0.021},
  {{profile_ex,looper,1},                1000,    0.000,   19.628}],
 { {profile_ex,looper,1},                1001,   61.542,   19.649},     %
 [{suspend,                                 1,   32.224,    0.000},
  {{erlang,integer_to_list,1},           1000,    9.505,    9.505},
  {garbage_collect,                         6,    0.164,    0.164},
  {{profile_ex,looper,1},                1000,    0.000,   19.628}]}.
```

这两个数加起来大约是29.2毫秒，非常接近looper进程的OWN时长。现在，两个进程的耗时都已经清楚了，二者之间的差异也凸显出来：funner进程中的fun函数在1000次调用中一共耗费了大约19毫秒，平均每次fun函数调用耗时19微秒。

回顾代码清单14-1中的代码，很容易看出相较于looper/1，funner/1更为耗时（除非编译器能够通过内联优化消去fun函数调用）。我们利用fprof证明了一件显而易见的事实，当然重点并不在此，而在于让你学会如何诠释这些文件并从中查出耗时瓶颈所在。

cprof和fprof为你的系统性能分析之旅奠定了良好的基础。它们既有助于评估代码中的问题，又可以在修复问题之后量化优化效果。

然而有些时候，即便定位了问题所在，代码中导致问题的原因也还是让人不明就里。在下一节中，我们将对Erlang语言的一些陷阱和缺陷进行讨论，从而在日常开发中予以规避。

14.3　Erlang 编程语言的缺陷

Erlang代码具有较为良好的可读性，其原因之一就在于语义简明[①]。大部分情况下，每个操作的成本都清晰可辨：没有隐式调用的对象构造函数和析构函数，没有运算符重载（因此+运算符绝不可能偷偷摸摸地复制整个对象），没有虚函数表带来的间接调用，没有临界区，也没有阻塞式的消息发送原语。当然，函数调用几乎是“无所不能”的，它们的行为并不一目了然，但通常每个函数都附有清晰的文档（至于那些你尚未研读过文档，或者压根儿就没有文档的函数，就要小心了）。

和任何编程语言一样，Erlang也不可避免地具有一些小小的缺陷。不妨先从Erlang的基本数据类型说起。谈到性能优化，总少不了它们。基本数据类型是语言中应用最多的部分，选用恰当的数据表现形式和数据处理方式可以令你事半功倍。

① 很多人对Erlang代码可读性良好的观点不以为然，甚至揶揄Erlang的语法有损视力（Erlang蹩脚的语法主要源自Prolog，相关历史可以参考“A History of Erlang”）。然而语法和语义是两回事，一旦适应了Erlang的语法和函数式编程风格，你便会发现相较于C++等语言来说，Erlang的语义的确更易于把握。——译者注

14.3.1 基本数据类型的性能特点

首先请注意，Erlang的数据类型的大小是以机器字（machine word）为单位来计算的。这是由BEAM模拟器的工作机制决定的。在32位机器上，一个字长4字节；在64位机器上，一个字长8字节。表14-1总结了各种基本数据类型的大小。

表14-1 Erlang数据类型的大小

数据类型	内存占用量
小整数（立即数，immediate）	1个字
大整数（大数，bignum）	至少3个字（可按需增长）
浮点数	在32位架构下占4个字，在64位架构下占3个字
原子	1个字（原子的名称字符串仅存在于Erlang节点的原子表中）
二进制串或位串	3至6个字+数据长度（以字为单位）
pid、端口或引用	本地进程/端口/引用占1个字，远程进程/端口/引用占5个字
fun函数	9至13个字+被闭包捕获的变量每个占1个字
元组	2个字+每个元素1个字
列表	1个字+每个元素两个字

让我们来看看这些数据类型在性能方面都有些什么特点。在以下的讨论中，fun函数可被视作带有额外元数据的元组，而pid（以及端口和引用）则与整数相似。

1. 小整数

小整数仅占一个字的内存，不过BEAM会将这个字中的若干比特位用作类型标签，以便区分不同的数据类型，如图14-2所示。

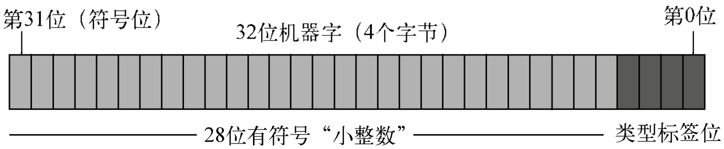


图14-2 BEAM中带类型标签的小整数。在32位机器上，可用于存储整数值的比特位只有28个。更大的整数必须表示成大数

在32位机器上，可用于存储整数值的比特位只有28个（包括符号位）；因此在单个字内，整数的取值范围为–134 217 728到+134 217 727，处理更大的整数时需换用大数（bignum）。

2. 大数

在Erlang中整数的大小不受限制。一个字长塞不下时，运行时系统会自动把它转换成长度可变的大数（但不可超出可用内存的大小）。二者之间唯一可感知的区别就是大整数运算比小整数运算要来得慢。在带有密集数值运算的紧凑循环中，如果给定的输入会导致大量大数运算，就会产生较为明显的性能差异。这时可以对程序进行修改，尽量使用小整数来完成运算。

3. 浮点数及其装箱形式

Erlang采用的是64位精度的浮点数，一个字长容纳不下（即便在64位机器上也放不下，和小整数的情形一样，BEAM会将一些比特位用作类型标签）。因此，浮点数必须表示成装箱形式：在这种形式下，浮点数的实际数据保存于进程的堆内存空间内，指向该位置的指针连同类型标签一并挤入一个字。这样一来，无论是用作函数参数还是用作数据结构的成员，需要该浮点数时只需复制这个字便可。接着来看位于堆上的数据，第一个字用于描述数据类别（浮点数）及数据长度。紧随其后的才是真正的64位浮点数：在32位机器上占两个字长，在64位机器上占一个字长。如图14-3所示。

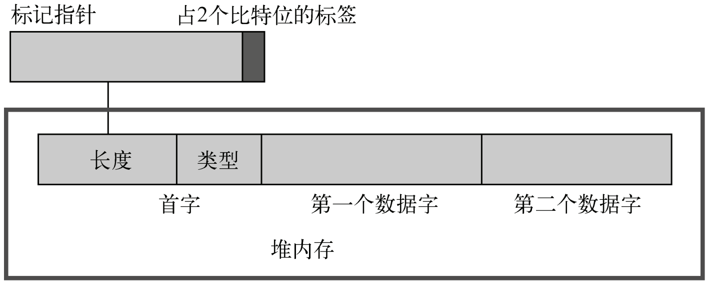


图14-3　浮点数和大数等一个字放不下的值的装箱形式。将这些值传给函数或插入别的数据结构时，只需传递标记指针

除浮点数外，还有几种基本数据类型也采用装箱形式，包括大数（这就是大数至少要占三个字长的原因）和元组。

4. 原子

原子的情形与小整数类似：每个原子只占一个字。原子的名称字符串保存在一张原子表中，每个Erlang节点只存一份。原子所占用的那个字中保存的实际上是原子表中对应名称字符串的索引。因此，原子的相等比较跟小整数的相等比较一样快[①]。由于效率高，原子被广泛用作标记元组的标签。模块加载时，模块中尚未加入表中的原子会被全部加入表中；此外，当前节点收到的发自其他节点的新原子，以及调用`list_to_atom(NameString)`产生的新原子，都会被写入原子表。然而原子不会被垃圾回收，插入表中的原子即便永不再使用也不会被删除，清理这张表的唯一途径就是重启节点。

动态创建原子会造成内存泄漏

出于种种目的，Erlang初学者往往会动态创建原子：`x1`、`x2`、…、`x187634`，诸如此类。对于那些一次性的、跑完就会关闭Erlang VM的程序来说，生成几百甚至几千个原子完全没问题。然而原子表的容量是有限的，目前只能容纳一百多万项。一旦溢出，VM便会报出“超出系统限制”（system limit）错误，然后崩溃。小程序一般不会超出这个限制，但对于需要长时

① 只需比较索引值是否相等。——译者注

间运行的线上系统来说这个问题却是致命的。

譬如，在将服务器接收到的外来数据转换成 Erlang 消息时就得特别小心。外来数据中的字符串应该转换为 Erlang 字符串或二进制串，要是转成了原子，那就等着中招吧：攻击者只需发送大量互不重复的字符串便可以把节点搞垮。

原子主要用于表示静态的标识符集合。需要多少用多少，但千万不要放任不可信的数据源随意生成原子。在将字符串转换为原子时，可以考虑使用BIF `list_to_existing_atom(NameString)`，它只会生成系统中已知的原子。倘若原子表中没有与字符串相对应的原子，该函数将抛出异常。

5. 二进制串和位串

二进制串和位串（参见2.2.2节）不过是些字节片段。它们的表现形式和大数类似，但却更为复杂，因为底层实际上存在若干种对上层不透明的不同类型的二进制串。它们主要分为两类：

- 堆型二进制串（较小）最大64字节。它们跟浮点数和大数一样，保存在进程自身的堆中。和其他Erlang数据类型一样，在进程间传递消息时，这类二进制串的数据会被一并复制。
- 引用计数型二进制串（较大）保存在单独的、为所有进程所共享的全局内存区域中，这些二进制串采用引用计数式垃圾回收。在同一VM内的多个进程之间传递这类大型二进制串时无须复制数据，只需传递一个指针即可。有了这一机制，我们便可以让一个进程从文件或端口中读取数据，再将读出的数据发送给另一个进程进行处理，完全不用担心数据复制的开销。通晓底层的实现机制固然是件好事，然而一味利用这些鲜为人知的特性盲目追求性能却绝非上策。

Erlang二进制串的语法很强大，也很容易用错，要做到运用自如绝非易事，在循环中处理二进制数据尤其困难。一个快速判断二进制串处理效率的方法就是启用`bin_opt_info`编译选项，譬如将系统环境变量`ERL_COMPILER_OPTIONS`设置成`[bin_opt_info]`。设置该选项后，编译器会专门针对代码中的二进制串输出一些颇有助益的警告和信息。

6. 元组

元组的情况就相当简单了。要知道元组是只读数据结构，更新就意味着复制。另外，记录实际上也是元组，所以更新记录字段就意味着创建新的元组：更新一个含有10个字段的记录，总共要写12个字。但另一方面，元组或记录中的字段选取操作却非常之快。简而言之，要么快速读取要么快速更新，鱼和熊掌不可兼得。对于恒定不变的数据，将大型元组用作数组可以提高访问效率，但更新效率堪忧。如果将元组嵌套成树状结构，虽然会引入多次间接寻址从而降低读取速度，但更新操作的效率却会得到提升。标准库中的array模块采用的就是这种做法。

7. 列表

列表的表现形式和编程注意事项前面已经讨论过了（参见2.2.5节、2.2.10节、2.15.5节和附录B），这里只说说列表的内存占用情况。回想一下先前探讨过的浮点数的装箱形式，列表单元也差不多，它与二元组类似，但在实现层面存有一个关键的差异点：列表单元（`list cell`）的第一个字包含一个特殊的类型标签和一个指针，其中标签表明这是一个列表单元，指针则指向其余的

位于堆上的数据。为了指明类型和元组的长度，二元组位于堆上的数据的最前端有一个用于保存这些附加信息的首部字；然而列表单元的元素数固定为两个，无须这些附加信息，只需堆上的两个字即可完整表示一个列表单元（参见2.2.10节中的图2-1和图2-2）。这一设计有效保障了用做通用数据结构的Erlang列表的效率。

字符串的内存占用量

字符串本质上就是字符构成的列表，字符的数值编码都是小整数，可以完整放入列表单元的第一个字内。因此字符串中的每个字符刚好占用两个字。将较短的字符串（比方说短于 10 000 个字符）用做临时数据结构不成问题。但如果将文本以字符串（即字符数值编码列表）形式存入数据库、ETS 表或别的数据结构中，就会造成空间的大量浪费。将字符串转换成二进制串可以将空间占用缩减至 1/8（在 64 位机器上是 1/16）。另一方面，也得考虑下这样做是否便利：如果内存充裕，当前空间占用根本不是问题，那么将代码中的字符串都换成二进制串可能并不值得，如果还要赔上代码的可读性，那就更加得不偿失了。

有关基本数据类型的讨论就到此为止了。下一节将对Erlang的BIF及运算符的性能问题展开讨论。

14.3.2　内置函数和运算符的性能

Erlang定义了大量运算符和BIF。作为运行时系统的一部分，它们都是用C写成的，一般来说效率不成问题，但我们仍然有必要考量它们的一些隐含的性能问题。在这一节中，我们将揭示与以下功能相关的几个常见的陷阱：

- `++`
- `--`
- `list_to_atom(CharList)`
- `length(List)`
- `size(Thing)`

1. ++运算符

我们曾经在在2.2.5节和2.15.5节的末尾讨论过这个问题，这里只是重复强调一遍：不要放任列表自右侧增长！此外还应注意++运算符只是`lists:append/2`的一个别名，这个问题对该函数同样适用。

2. --运算符

`--`运算符是`lists:subtract/2`的别名。这个运算符并不常用：它的作用是从左侧列表中删除右侧列表中的元素。然而，针对右侧列表中的每一个元素，`--`运算符只会删除左侧列表中自左往右出现的第一个匹配项。因此要想完全删除指定的元素，就必须知道该元素在左侧列表中的出现次数。参见以下示例：

```
1> [1,2,3,2,1] -- [2,1,2].
[3,1]
```

可见，被删除的元素包括左侧的第一个1和两个2，3则不受影响。元素的次序也不受影响。

这个函数的缺点在于它用的是平方复杂度的算法：针对右侧列表中的每个元素，都要对左侧列表做一次遍历。列表较短时表现并不明显；但一旦碰上长列表，就很成问题了。如果元素的次序并不重要，那么更为高效的做法是先排序，再调用`ordsets:subtract/2`。

3. `list_to_atom/1`

切记原子无法被垃圾回收，务必小心使用这个函数。在某些应用场景下也许`list_to_existing_atom/1`会更为合适，详情参见14.3.1节中和原子相关的讨论。

4. `length/1`

请记住为了计算元素数目，`length(List)`必须遍历整张列表，这点与计算字符串长度的C函数`strlen`很相似。有Java开发背景的程序员往往会忽略这点，以为这是个常数复杂度的操作——非也！在若干常见场景下可以利用模式匹配来替代`length/1`，详情请参见2.15.5节的末尾。

5. `size/1`

`length/1`和`size/1`的适用范围很容易混淆：`length/1`仅适用于列表，`size/1`同时适用于元组和二进制串，但不适用于列表。二者之间的区别在于`size/1`是常数操作，无须遍历所有数据，因此非常迅捷。然而`size/1`函数既适用于元组又适用于二进制串这一点却又会导致一些小小的麻烦，因为该函数用于前者时返回的是元素数，用于后者时返回的却是字节数。光看代码，我们很难分辨参数中给定的是元组还是二进制串，搞不清这一点，就更搞不清函数返回的数值究竟代表什么意义了。这种情况会让Dialyzer等代码检查工具抓瞎，难以发现潜在的错误。

在现代Erlang代码里，我们推荐用`tuple_size(T)`获取元组的元素数，用`byte_size(B)`或`bit_size(B)`获取二进制串或位串中的字节数或比特位数。对于（长度不可被8整除的）位串，`byte_size(B)`会向上取整——也就是说，它返回的是能容纳`B`中所有比特位的最小字节数。使用这些函数可以清晰地表明意图，而且可以为编译器和Dialyzer工具提供关键线索。

现在BIF和运算符也讨论完了，下面我们来说说普通的用户自定义函数的效率。

14.3.3 函数

在必要的情况下，有效运用函数可以再“榨出”几毫秒。在这一方面，你可能会有些困惑和误解。我们不妨从表14-2开始，这张表总结了各种函数调用方式的耗时情况。

表14-2 函数调用速度

函数调用类型	耗　　时
本地函数：`foo()`	非常快
已知的远程函数：`bar:foo()`	几乎和本地函数调用一样快
未知的远程函数：`Mod:Func()`	大约比本地调用慢3倍
Fun函数调用：`F()`	比本地调用慢2 ~ 3倍
元调用：`apply(Mod,Func,Args)`	比本地调用慢6 ~ 10倍

14

绝对耗时取决于硬件速度，相对耗时也会随编译器和运行时系统的版本而变化。例如，（在多年以前）调用其他模块中的函数比调用本地函数要慢得多。现如今，二者已经差不多了。从表14-2可以看出，除非是对性能要求极其苛刻的代码，否则一般情况下无须太过关注函数调用的开销：只有元调用的速度显著落后，不过它本来就不常用。请记住，在参数数目固定的情况下，Mod:Fun(...)形式优于apply/3。

1. 尾递归和非尾递归

在2.15.2节中我们曾经讨论过尾递归和非尾递归。它们之间的关系类似于本地调用与远程调用的关系，一般情况下非尾递归比尾递归要慢。归功于运行时系统和编译器的进步，如今二者之间的差异已经很小了，较为优雅的非尾递归函数在效率上跟尾递归版本往往相差无几。

在速度至关重要的情况下，如果有精力把尾递归版本和非尾递归版本各实现一遍（依具体问题而不同，其中一个版本实现起来会更为容易），就不要妄作假设——对比两个版本的评测数据，让数字说话。受缓存实现策略等因素的影响，不同硬件平台下的结果可能会有所不同。此外，输入数据的规模也会产生显著影响（输入规模决定递归深度），小规模输入和大规模输入都应予以评测；大多数情况下输入规模如何，最差情况和平均情况下的耗时是否关键，这些都得考虑清楚。

2. 子句选择

碰到含有多个子句的函数或fun表达式（也包括case、if、try或receive表达式）时，为了快速选出合适的子句，编译器会尽量削减判断次数。这套名为*模式匹配编译*的算法会将子句分组排序，再切分成一系列嵌套的if/then/else判断。然而它只会调整判断的执行次序，（除提速以外）不会影响输出。

譬如，判断输入是原子true还是false时，判断次序无关紧要，因为二者是互斥的：它们不可能同时成立。类似的判断还包括检验列表是否为空等。然而某些情况下多个选项会相互覆盖，以下列函数为例

```
coffee_size(N) when N < 12 -> short;
coffee_size(N) when N < 16 -> tall;
coffee_size(N) when N < 20 -> grande;
coffee_size(_)             -> venti.
```

函数的逻辑受制于判断的次序，如果对调最上方的两个子句，包括所有小于12的N在内，所有小于16的N都会得出tall。编译器只会调整那些安全无虞的子句的次序，其余内容则保留原样。

有一点需要注意的是，不要在简明的子句之间插入带有不确定性的子句。以下列代码为例：

```
handle_message(stop) -> do_stop();
handle_message(go)   -> do_go();
handle_message(Msg) when Msg =:= Special -> do_special(Msg);
handle_message(report)    -> do_report();
handle_message(calc)      -> do_calculate();
handle_message(OtherMsg) -> do_error(Other).
```

该函数的输入应是指定的4个原子之一，或是某个特殊项式Special（一个运行时确定的变量），遇到其他项式则报错。运行时的Special有可能同为原子，但编译器无从知晓。尤其是，运行时的Special有可能就是原子report或calc二者之一，这样一来，stop、go、report和calc4个子句就不能归并在一起。编译器最多也就是把stop和go判断归并为一组，接着判断Special，再将report和calc归并为一组。身为开发者，如果你确认Special的取值永远不会是report或calc，那么就应该把它挪到另外两个判断的下方。仅就这个示例而言，性能上的差异非常小；但要是碰上很多带有复杂模式的子句，对各个判断进行恰当的分组归并还是值得的。（这样做还会提升可读性，令代码更为清晰地表达开发者的意图。）

在本章的最后，我们来探讨一下进程的效率问题。

14.3.4 进程

进程是所有Erlang程序的基本执行环境。所有代码都要依托于进程才能执行。即便是自身不启动任何进程的库模块的代码，运行时也要依托于调用它的进程才行。

如前所述，Erlang中的进程十分“廉价”。大量进程并发运行在Erlang中可谓司空见惯。然而每个进程执行的工作却会对整个系统的性能产生显著影响。

1. 要不要用OTP行为模式

虽然新进程的创建仅需数毫秒，但OTP行为模式容器进程的初始化却是另外一回事。gen_server:start_link()调用会引发一系列动作，包括调用行为模式实现模块中的init/1回调。前文曾经提过，init/1回调不结束，start_link函数就不会返回。这一设计是为了保证服务启动过程的确定性，确保当调用方拿到新服务器进程的ID时，服务器已经完成了初始化并且随时可以接受请求。

在某些场景下，大量进程来去匆匆。以第11章中的连接处理进程为例，代码清单11-3中的ti_server便是如此。这类服务会为每个TCP连接派生一个新的进程。在大压力下，测试数据表明大量时间被耗费在进程初始化上。进程的生存期越短，耗费在OTP库代码上的时间比就越高。在速度至上的情况下，抛弃OTP行为模式，转而直接利用spawn自行打造轻量级的进程管理机制也许更为实际，这样做可以在最大程度缩减开销的同时提供精简的控制功能。然而这种做法很容易出错，只可用于处理非常情况，而且只有在熟练掌握进程和OTP编程之后才行，这样你才会明白自己为了性能而放弃了些什么。

2. 设置堆的初始尺寸

如果大量进程在创建之后快速消亡，那么还可以采取另外一种优化措施：调大每个进程的初始堆大小，以避免垃圾回收及进程启动之后的内存分配。

进程堆的默认大小是233个字（在32位机器上等于932字节），后续还可按需增减。这一自动内存管理机制十分方便，但会带来一定的运行时开销。如果能够算出这些临时进程在它们短暂的生命周期内总共需要多少内存，就可以在启动它们时预先设置堆的初始大小。这一任务可借由spawn_opt系列函数完成：

```
erlang:spawn_opt(Fun, [{min_heap_size, Words}])
```

在这种方式下，每个进程都被视作一块内存区域，这块内存在进程启动时分配，在进程结束时回收，除此之外不再需要其他内存管理。如图14-4所示。

这么做的缺点在于每个进程的内存占用量都高于实际需要，因此实际上是在拿内存空间换速度。

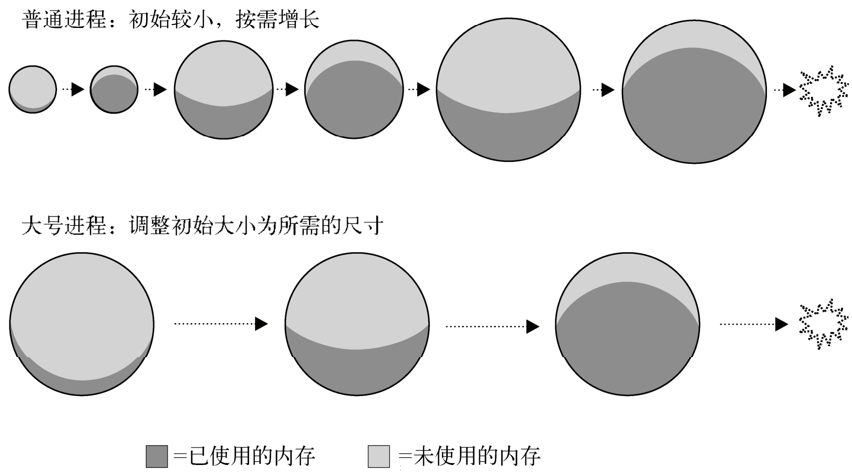


图14-4　为生存期较短而工作量较为固定的进程设置更大的初始堆，以避免垃圾回收和堆大小的调整

3. 休眠

如果有大量进程需要长期保持活跃，且其中大部分进程因等待消息而处于睡眠状态，就可以考虑让这些进程转入休眠状态。

调用`erlang:hibernate(Mode, Func, Args)`即可令进程休眠。休眠的进程会抛弃调用栈，忘却自身在程序中的当前执行位置。有鉴于此，`hibernate/3`永不返回，当前正活跃的`catch`或`try/catch`表达式也会被忽略。接着，将会强制执行一次垃圾回收，精简进程的内存占用。最后，进程进入睡眠状态，直到新消息再次进入信箱。（若休眠时信箱不为空，进程将被立即唤醒。）进程被唤醒后的行为就仿佛是调用了`apply(Mod,Func,Args)`，不过该“调用”没有返回地址。

休眠可以精简睡眠中的进程的内存占用，释放出更多的空间容纳更多的进程。这一手法特别适用于那些监控着大量外部实体的系统。

基于proc_lib的进程，如gen_server及其他OTP行为模式，应该使用`proc_lib:hibernate/3`而不是`erlang:hibernate/3`，以确保进程醒来后周遭一切都遵照OTP库的约定再次打点妥当。

好了，性能相关的讨论就到此结束了。

14.4　小结

在这一章中，我们讨论了性能调优的基本方法：设定目标、度量系统行为、确定亟待解决的问题，并确认修改是否行之有效。你学会了如何使用cprof和fprof工具来分析代码的执行时间。我们还介绍了Erlang代码中和效率相关的各种缺陷、陷阱和技巧。掌握了这些知识，你便具备了参与Erlware团队性能优化工作的能力。记住，代码的执行效率固然重要，但千万不要忘了代码之美才是最值得追求的。

附录 A

安装Erlang

随操作系统和个人喜好的不同，Erlang的安装方法也各不相同。当前，Erlang在新版Windows、Mac OS X、Linux、大部分类UNIX操作系统，以及VxWorks实时操作系统上都可以运行。

A.1 在Windows上安装Erlang

使用Windows操作系统的用户，从www.erlang.org/download.html下载并运行最新版本的.exe安装文件即可。安装文件中囊括了文档，还会设置好开始菜单和Windows专用的特殊shell werl的快捷方式（参见2.1.1节）。

A.2 在Mac OS X、Linux等类UNIX系统上安装Erlang

在类UNIX系统下，如果需要最新版本，可以通过源码编译安装Erlang；在某些系统下，可以借助软件包管理器（如Ubuntu的synaptic）下载安装对应系统下的官方Erlang软件包——但这样安装的Erlang可能并非最新版本。在Mac OS X下，可通过Homebrew软件包管理器（http://mxcl.github.com/homebrew）获取当前版本的Erlang。

A.2.1 编译源码

通过浏览器访问www.erlang.org/download.html并获取最新版本的源码包。下载完毕后，解开tar包，cd进入目录，然后执行./configure脚本。默认安装位置是/usr/local/lib/erlang，如果不打算安装到这个位置下，可以加上`--prefix=...`参数。譬如：

```
./configure --prefix=/home/jdoe/lib
```

确认configure执行无误后（如有问题请参见下一小节），执行

```
make
```

再执行

```
make install
```

请注意，如若安装至默认位置，安装过程可能需要root权限。在当今大多数系统下，执行以下

命令即可

```
sudo make install
```

安装完毕后，便可用`erl`启动`Erlang`或者用`erlc`调用编译器。如果采用的是非标准安装路径，请确保相应路径下的子目录已列于`PATH`环境变量内。

A.2.2　解决配置问题

编译Erlang源码时，系统中必须事先安装一些库和工具。其中系统默认情况下可能没有的常见软件包包括：

- 完整的GCC编译器环境；
- Ncurses开发库。

装好缺失的软件包后，在运行make之前，请再次运行configure（软件包的具体名称可能会随系统的不同而不同）。

有些库并非必需，如有缺失，配置步骤只会给出警告，告诉你某些Erlang应用将被禁用。如果原本就用不上这些应用，大可直接运行`make`；否则，你还得安装缺失的库并重新运行configure。常见的几个库包括：

- OpenSSL开发库
- ODBC开发库
- Java

倘若禁用了某些应用，回头却又发现用得上它们，可以再次运行./configure、make、make install来安装缺失的软件包。

附录 B

列表与引用透明性

你可能会问，Erlang数据类型中的列表究竟有什么*用意*呢（尤其是习惯了Java、Python等语言中的数组、缓冲区等概念之后）？每个元素都要附带一个指向下一个元素的指针，不仅浪费内存，而且还不能直接从列表的右侧添加元素！哼！

这些都没错，但在Erlang中，变量的值不容修改，任何人的数据都不会被偷偷摸摸的篡改。这便是*引用透明性*这一术语背后的含义。

B.1 引用透明性的定义

引用透明性的概念很简单。简而言之就是：为了跟踪某个值（*项式*），我们赋予它一个名字（如X），从此往后，X的引用可以被传递给程序的任意部分，但X指代的值却恒定不变。换言之，变量所指代的值（或值的一部分）绝不会悄无声息地变动。不难看出，Erlang的单次赋值变量正是这一原则的集中体现。

B.2 引用透明性的优点

Java中的字符串之所以是常量，背后也是同样的原因：假设你原本打算打印的字符串是："神父，还要茶吗？"结果却因为在打印之前将字符串的引用交给了某个不守规矩的函数，最终打印出一句粗暴无礼的话，那该多尴尬呀！

言归正传，之所以Erlang中的数据都要遵循引用透明性，原因如下。

- 大幅减少程序错误——对于长达百万行代码、涉及数十甚至数百程序员的大型项目来说，这一点非常重要。
- 将单进程模式下运作良好的代码切分至多个进程时，无须重写代码：在将代码切分至多个进程（可能运行于多台机器上）时，不必担心碰到会导致代码无法正常工作的陷阱。
- 由于不存在对现有数据结构的写操作，系统可以对内存管理和多线程管理进行更具创新性的优化。

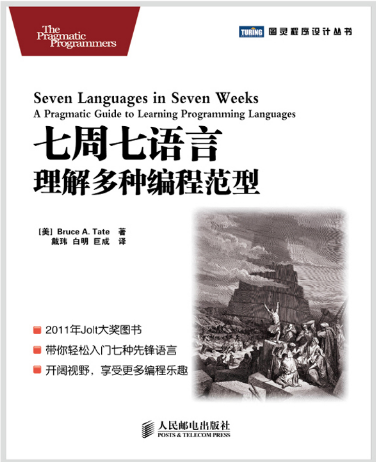
因此，引用透明性绝不仅仅是身处象牙塔内的少数人的理论追求——它对程序的稳定性、可扩展性有着深远的影响；更不用说对代码的可读性、可调试性，以及开发速度所起到的积极作用了。

B.3　与列表的关系

回到列表上来，要保障手头现有的列表（可能是来自函数的参数）的引用透明性，就意味着不能向列表的末尾追加元素。否则同样持有该列表的引用的人会发现列表的末尾无中生有地多出了一个元素。这在Erlang中是不允许的。

然而从左侧添加元素却没有问题（利用列表单元），因为原列表不会受到影响——只需创建一个新的列表单元，并指向原列表的第一个单元，这就好像是在说："这个列表不错，我要了，不过表头还得追加一个新元素。"

因此，列表单元可以优雅地解决引用透明系统中列表的动态增长问题；而且列表单元的实现方案也非常简单高效。（几十年来）许多聪明绝顶的人士都试图寻找这样一种解决方案——在保留引用透明特性的同时，既可以支持在末尾追加元素又可以保持较低的内存消耗，但最终这些方案实现起来都过于复杂，在通用情况下效率也不理想。



本书荣获2011年Jolt大奖，深度挖掘以下七种语言的精华：

- Ruby
- Io
- Prolog
- Scala
- Erlang
- Clojure
- Haskell

“惊艳！不管是对于初学者还是Erlang高手，本书绝对都是不容错过的好书。” ——Amazon.com书评

“多核处理器和并发编程是将来的重头戏，Erlang在下一代编程语言中可谓独领风骚！” ——DZone书评

“Erlang开发者必备两本书，一本是Erlang之父Joe Armstrong的《Erlang程序设计》，另一本就是本书——务实、高效又不失幽默风趣的好书啊！” ——slashdot.org书评

Erlang and OTP in Action

Erlang/OTP并发编程实战

通过提高CPU时钟频率来制造更快的单核芯片的技术已经到达了极限。多核、并发、分布式等概念和技术也随之走出象牙塔，成为每个一线开发者的必备技能。由通信巨头爱立信研发的Erlang/OTP大放异彩，二十多年来，在传统电信领域高并发、高可靠、高容错的严酷环境下，Erlang语言和OTP平台被锻炼得坚如磐石，浓郁的函数式特质更是恰到好处地弥补了传统命令式语言在并发编程上的固有缺陷，大大降低了构筑并发、容错、分布式应用的门槛。

如果将Erlang语言看成才华横溢的钢琴家，那么OTP平台就是一架能让钢琴家把才能发挥得淋漓尽致的钢琴。本书除了全面介绍Erlang语言和OTP平台的基础知识外，还通过一系列实用案例引领你深入了解OTP的高级特性，一步步构建一个大型生产系统，并加以优化和完善。三位作者在Erlang领域拥有极其丰富的实战经验，细致入微地剖析了OTP开发与部署的全过程。要想真刀真枪地上战场，本书才是你明智的选择！

- 首部OTP开发部署实战指南
- 着眼于产品级代码开发
- 各级Erlang开发人员必备读物

MANNING

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育　@图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议　计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-28559-1
定价：79.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn